

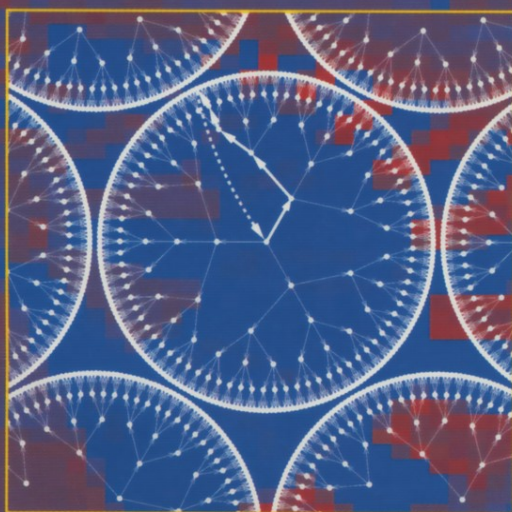
State-of-the-Art  
Survey

LNC3 3016

Christian Lengauer  
Don Batory  
Charles Consel  
Martin Odersky (Eds.)

# Domain-Specific Program Generation

**International Seminar**  
**Dagstuhl Castle, Germany, March 2003**  
**Revised Papers**



Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*University of Dortmund, Germany*

Madhu Sudan

*Massachusetts Institute of Technology, MA, USA*

Demetri Terzopoulos

*New York University, NY, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Moshe Y. Vardi

*Rice University, Houston, TX, USA*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Christian Lengauer Don Batory  
Charles Consel Martin Odersky (Eds.)

# Domain-Specific Program Generation

International Seminar  
Dagstuhl Castle, Germany, March 23-28, 2003  
Revised Papers

## Volume Editors

Christian Lengauer  
Universität Passau, Fakultät für Mathematik und Informatik  
94030 Passau, Germany  
E-mail: lengauer@fmi.uni-passau.de

Don Batory  
The University of Texas at Austin, Department of Computer Sciences  
Austin, TX 78712, USA  
E-mail: batory@cs.utexas.edu

Charles Consel  
INRIA/LaBRI, ENSEIRB, 1, avenue du docteur Albert Schweitzer  
Domaine universitaire, BP 99, 33402 Talence Cedex, France  
E-mail: consel@labri.fr

Martin Odersky  
École Polytechnique Fédérale de Lausanne (EPFL)  
1015 Lausanne, Switzerland  
E-mail: martin.odersky@epfl.ch

Library of Congress Control Number: 2004105982

CR Subject Classification (1998): D.1, D.2, D.3

ISSN 0302-9743

ISBN 3-540-22119-0 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable to prosecution under the German Copyright Law.

Springer-Verlag is a part of Springer Science+Business Media  
springeronline.com

© Springer-Verlag Berlin Heidelberg 2004  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik  
Printed on acid-free paper      SPIN: 11011682      06/3142      5 4 3 2 1 0

# Preface

One of the goals of software production in future decades must be to reduce production costs and time to market considerably, while improving the quality and stability of the product. This can be achieved most effectively if the broad mass of application programmers is given the opportunity to apply their expert knowledge in their application domain but is not expected to have in-depth knowledge of the issues of software engineering and implementation.

The considerable gap between application-level problem solutions and efficient implementations at the level of today's source programs as written in C or Java must be bridged by sophisticated optimizing techniques of program generation. At best, these techniques should be fully automatable – at least, they should have strong automatic support. If this can be achieved, program generation has the potential to revolutionize software development just like automation and components revolutionized manufacturing.

This book is about domain-specific program generation. It is a condensation of contributions all but one of which were presented at a five-day seminar with the same title at Schloss Dagstuhl, Germany, in March of 2003. The seminar brought together researchers from four different communities:

- *Domain-specific languages*: Language developers in a specific application domain have often been unaware of the domain-independent aspects of their domain-specific work. Vice versa, researchers who do not work in a specific domain are often unaware of some of the factors that make an application work.
- *High-performance parallelism*: This is one application domain which has led to the development of a particular form of domain-specific language (so-called skeletons). Researchers in this community are becoming interested in the wider aspects of domain-specific program generation.
- *Program generators*: This domain is concerned with the fast and reliable generation of members of a program family, also called a product line. There are many applications of product lines in industry, commerce, and the military.
- *Metaprogramming*: Researchers in this community develop a technology for combining and specializing program fragments. This requires at least two levels of code: one in which the fragments are coded and one which combines and specializes the fragments. This technology can be used for customizing compilation and translation systems for domain-specific purposes. Multi-staging is a special form of metaprogramming, in which each level is coded in the same programming language.

This volume has four parts.

**Surveys.** Six surveys attempt to give some structure to the diverse world of domain-specific programming technology.

1. Batory retraces the evolution of the very successful domain of data base query optimization and discusses what lessons can potentially be learned for other domains.

2. Consel makes the point that a domain is best defined as a set of existing programs and sketches how one might derive a domain-specific language from such a set, with which one can then specify other programs in this set.
3. Taha illustrates the technique of multi-stage programming on the example of a staged interpreter.
4. Czarnecki et al. describe staged interpreters and templates as a suitable way of extending a host language with domain-specific constructs. They evaluate and compare three languages for template programming: MetaOCaml, Template Haskell and C++.
5. One step beyond domain-specific program generation lies the goal of domain-specific program optimization. Lengauer reviews different optimization techniques in the domain of high-performance parallelism.
6. Smaragdakis offers a personal assessment of the approaches and attitudes in the research community of generative programming.

**Domain-Specific Languages.** Five contributions describe domain-specific programming languages or language enhancements.

1. Bischof, Gorlatch and Leshchinskiy present the skeleton library DatTeL for the domain of high-performance parallelism. The library's two key features are (1) its user interface, which is similar to the C++ Standard Template Library (STL) and which facilitates a smooth transition from sequential to parallel programming, and (2) an efficient implementation of STL-like constructs on parallel computers.
2. Hammond and Michaelson describe the language Hume for the domain of real-time embedded systems. Hume has high-level features typical for functional languages. Since it consists of three layers, Hume also allows for domain-specific metaprogramming.
3. O'Donnell describes an embedding of the language Hydra for the domain of digital circuit design into the host language Template Haskell.
4. Consel and Réveillère present a programming paradigm for the domain of services for mobile communication terminals. This domain is subject to frequent changes in technology and user requirements. The paradigm enables the quick development of robust communication services under these challenging circumstances.
5. Cremet and Odersky choose the  $\pi$ -calculus for mobile processes as their domain. They describe the domain-specific language PiLIB, which is implemented as a library in Odersky's new language SCALA. With the features of SCALA, calls to PiLIB can be made to look almost like  $\pi$ -formulas.

**Tools for Program Generation.** Three further contributions stress issues of tool support for program generation in a domain-specific language.

1. Gregg and Ertl work with their language vmlDL for describing virtual machine instructions. Their tool vmgen takes the specification of such instructions in their domain-specific language and returns efficient implementations of the instructions in C.

2. Visser presents the Stratego language for the domain of rewriting program transformations, and the corresponding toolset Stratego/XT.
3. Fischer and Visser work with AUTOBAYES, a fully automatic, schema-based program synthesis system for applications in the analysis of statistical data. It consists of a domain-specific schema library, implemented in Prolog. In their contribution to this volume, they discuss the software engineering challenges in retro-fitting the system with a concrete, domain-specific syntax.

**Domain-Specific Optimization.** Finally, four contributions describe domain-specific techniques of program optimization.

1. Kuchen works with a skeleton library for high-performance parallelism, similar to the library DatTel of Bischof et al. However, Kuchen's library is not based on STL. Instead, it contains alternative C++ templates which are higher-order, enable type polymorphism and allow for partial application. In the second half of his treatise, Kuchen discusses ways of optimizing (i.e., "retuning") sequences of calls of his skeletons.
2. Gorlatch addresses exactly the same problem: the optimization of sequences of skeleton calls. His skeletons are basic patterns of communication and computation – so-called collective operations, some of which are found in standard communication libraries like MPI. Gorlatch also discusses how to tune compositions of skeleton calls for a distributed execution on the Grid.
3. Beckmann et al. describe the TaskGraph library: a further library for C++ which can be used to optimize, restructure and specialize parallel target code at run time. They demonstrate that the effort spent on the context-sensitive optimization can be heavily outweighed by the gains in performance.
4. Veldhuizen pursues the idea of a compiler for an extensible language, which can give formal guarantees of the performance of its target code.

Each submission was reviewed by one person who was present at the corresponding presentation at Schloss Dagstuhl and one person who did not attend the Dagstuhl seminar.<sup>1</sup> There were two rounds of reviews. Aside from the editors themselves, the reviewers were:

Ira Baxter	Christoph M. Kirsch	Ulrik Schultz
Claus Braband	Shriram Krishnamurthy	Tim Sheard
Krzysztof Czarnecki	Calvin Lin	Satnam Singh
Albert Cohen	Andrew Lumsdaine	Yannis Smaragdakis
Marco Danelutto	Anne-Françoise Le Meur	Jrg Striegnitz
Olivier Danvy	Jim Neighbors	Andrew Tolmach
Prem Devanbu	John O'Donnell	Todd Veldhuizen
Sergei Gorlatch	Catuscia Palamidessi	Harrick Vin
Kevin Hammond	Susanna Pelagatti	David Wile
Christoph A. Herrmann	Simon Peyton Jones	Matthias Zenger
Zhenjiang Hu	Frank Pfenning	
Paul H. J. Kelly	Laurent Réveillère	

<sup>1</sup> An exception is the contribution by Cremet and Odersky, which was not presented at Schloss Dagstuhl.

**IFIP Working Group.** At the Dagstuhl seminar, plans were made to form an IFIP TC-2 Working Group: WG 2.11 on Program Generation. In the meantime, IFIP has given permission to go ahead with the formation. The mission statement of the group follows this preface.

**Acknowledgements.** The editors, who were also the organizers of the Dagstuhl seminar, would like to thank the participants of the seminar for their contributions and the reviewers for their thoughtful reviews and rereviews. The first editor is grateful to Johanna Bucur for her help in the final preparation of the book.

We hope that this volume will be a good ambassador for the new IFIP WG.

March 2004

Christian Lengauer  
Don Batory  
Charles Consel  
Martin Odersky



# IFIP WG 2.11 on Program Generation

## Mission Statement

### Aim

Generative approaches have the potential to revolutionize software development as automation and components revolutionized manufacturing. At the same time, the abundance of current research in this area indicates that there is a host of technical problems both at the foundational and at the engineering level. As such, the aim of this working group of researchers and practitioners is to promote progress in this area.

### Scope

The scope of this WG includes the design, analysis, generation, and quality control of generative programs and the programs that they generate.

Specific research themes include (but are not limited to) the following areas:

- Foundations: language design, semantics, type systems, formal methods, multi-stage and multi-level languages, validation and verification.
- Design: models of generative programming, domain engineering, domain analysis and design, system family and product line engineering, model-driven development, separation of concerns, aspect-oriented modeling, feature-oriented modeling.
- Engineering: practices in the context of program generation, such as requirements elicitation and management, software process engineering and management, software maintenance, software estimation and measurement.
- Techniques: meta-programming, staging, templates, inlining, macro expansion, reflection, partial evaluation, intentional programming, stepwise refinement, software reuse, adaptive compilation, runtime code generation, compilation, integration of domain-specific languages, testing.
- Tools: open compilers, extensible programming environments, active libraries, frame processors, program transformation systems, program specializers, aspect weavers, tools for domain modeling.
- Applications: IT infrastructure, finance, telecom, automotive, aerospace, space applications, scientific computing, health, life sciences, manufacturing, government, systems software and middle-ware, embedded and real-time systems, generation of non-code artifacts.

### Objectives

- Foster collaboration and interaction between researchers from domain engineering and those working on language design, meta-programming techniques, and generative methodologies.
- Demonstrate concrete benefits in specific application areas.
- Develop techniques to assess productivity, reliability, and usability.

# Table of Contents

## Surveys

The Road to Utopia: A Future for Generative Programming . . . . .	1
<i>Don Batory</i>	
From a Program Family to a Domain-Specific Language . . . . .	19
<i>Charles Consel</i>	
A Gentle Introduction to Multi-stage Programming . . . . .	30
<i>Walid Taha</i>	
DSL Implementation in MetaOCaml, Template Haskell, and C++ . . . . .	51
<i>Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha</i>	
Program Optimization in the Domain of High-Performance Parallelism . . .	73
<i>Christian Lengauer</i>	
A Personal Outlook on Generator Research (A Position Paper) . . . . .	92
<i>Yannis Smaragdakis</i>	

## Domain-Specific Languages

Generic Parallel Programming Using C++ Templates and Skeletons . . . . .	107
<i>Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy</i>	
The Design of Hume: A High-Level Language for the Real-Time Embedded Systems Domain . . . . .	127
<i>Kevin Hammond and Greg Michaelson</i>	
Embedding a Hardware Description Language in Template Haskell . . . . .	143
<i>John T. O'Donnell</i>	
A DSL Paradigm for Domains of Services: A Study of Communication Services . . . . .	165
<i>Charles Consel and Laurent Réveillère</i>	
PiLIB: A Hosted Language for Pi-Calculus Style Concurrency . . . . .	180
<i>Vincent Cremet and Martin Odersky</i>	

## Tools for Program Generation

A Language and Tool for Generating Efficient Virtual Machine Interpreters . . . . .	196
<i>David Gregg and M. Anton Ertl</i>	

Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9 .....	216
<i>Eelco Visser</i>	
Retrofitting the AutoBayes Program Synthesis System with Concrete Syntax .....	239
<i>Bernd Fischer and Eelco Visser</i>	
<b>Domain-Specific Optimization</b>	
Optimizing Sequences of Skeleton Calls .....	254
<i>Herbert Kuchen</i>	
Domain-Specific Optimizations of Composed Parallel Components .....	274
<i>Sergei Gorlatch</i>	
Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation .....	291
<i>Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H.J. Kelly</i>	
Guaranteed Optimization for Domain-Specific Programming .....	307
<i>Todd L. Veldhuizen</i>	
<b>Author Index</b> .....	325

# The Road to Utopia: A Future for Generative Programming\*

Don Batory

Department of Computer Sciences  
University of Texas at Austin  
Austin, Texas 78712  
batory@cs.utexas.edu

**Abstract.** The future of software engineering lies in automation and will exploit the combined strengths of generative programming, domain-specific languages, and automatic programming. While each of these areas is still in its infancy, a spectacularly successful example of their combination was realized twenty-five years ago: relational query optimization. In this paper, I chart the successes and mindset used by database researchers to generate efficient query processing programs automatically. I argue that the road that they have so successfully followed is the same road that the generative programming, domain-specific languages, and automatic programming communities are now traversing.

## 1 Introduction

Just as the structure of matter is fundamental to chemistry and physics, so too is the structure of software fundamental to computer science. By the term ‘structure’ I mean what are modules, and how are modules composed to build programs? Unfortunately, the structure of software is not well-understood. *Software design*, which is the process by which the structure of an application is defined, is an art form. And as long as it remains so, our abilities to automate software development and make software engineering a true engineering discipline will be limited.

Our goal should be to create a mathematical science of software design. That is, we need to create general purpose theories of how customized software can be synthesized automatically. Object-oriented models are adequate if we implement programs manually; higher-level representations of programs are required for program synthesis. These theories will embody advances in *generative programming* (GP). That is, we want to understand the programs in a domain so well that they can be generated automatically. We want generators to synthesize these programs and do the hard technical work for us. This is a shared goal

---

\* Author’s note: This is the text of a keynote presentation at the Dagstuhl Seminar for Domain-Specific Program Generation, March 2003. The quotations within this paper are taken from the pre-seminar surveys that invitees identified as key research issues; the quotations from Jim Neighbors are from his review of this paper.

of the generative programming, metaprogramming, and skeleton communities. Program generation should *not* be an ad-hoc set of implementation techniques; rather, it is essential that we develop practical theories that integrate programming concepts and domain knowledge to automate software development.

We also need advances in *domain-specific languages (DSLs)*, which are special-purpose programming languages (or extensions to general-purpose languages) that allow programmers to more easily express programs in terms of domain-specific abstractions (e.g., state machines, EJB declarations). We do not want to be programming in Java and C# twenty years from now. Rather, we want to elevate program specifications to compact domain-specific notations that are easier to write, understand, and maintain.

And finally, we need advances in *automatic programming (AP)*. This is the extreme of GP and DSLs. Namely, the challenge of AP is to synthesize an efficient program from a declarative specification. This is a very hard problem; in the early 1980s, researchers abandoned AP as existing techniques simply did not scale to programs beyond a few hundred lines [1]. Now AP is undergoing a renaissance, and its need (e.g., for fault tolerance) is even more critical than ever.

To make advances simultaneously on *all* these fronts seems impossible. Yet, there exists a spectacular example of GP, DSLs, and AP in a fundamental area of computer science. And ironically, it was achieved about the same time that others were giving up on AP. The area is databases; the result is relational query optimizers.

In this paper, I review the successes and mindset that database researchers used to generate efficient query processing programs automatically and explain that the road that they followed so successfully is the same road that the GP, DSL, and AP communities are now traversing. I cite lessons that should be learned and chart a road-map that others could follow to achieve comparable successes in other domains. I use “Utopia” as the name of the objective that lies at the end of this road.

## 2 Lessons Learned and Lessons to Be Learned

### 2.1 Relational Query Processing

Relational queries are expressed as SQL SELECT statements. A parser translates a SELECT statement into an inefficient relational algebra expression, a query optimizer rewrites this expression into an equivalent expression that has better (or optimal) performance properties, and a code generator translates the optimized expression into an executable program (Figure 1).

SQL is a classic example of a declarative DSL. It is a language that is specific to tabular representations of data. The code generator, which maps a relational algebra expression to an executable program, is an example of GP. The query optimizer is the key to AP: it searches the space of semantically equivalent expressions to locate an expression which has good (or optimal) performance characteristics.

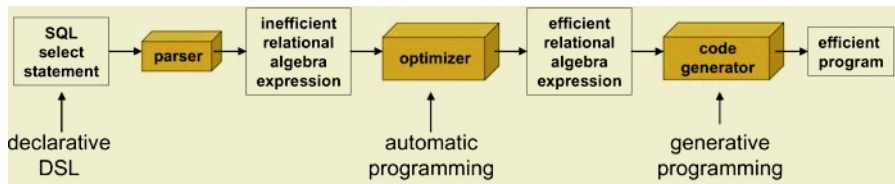


Fig. 1. Relational Query Optimization

Relational query processing is an instance of a very powerful and very successful paradigm that researchers in GP should strive to replicate in other domains. If this could be achieved, would this be Utopia? No, but it would be on the road...

## 2.2 Cognitive Systems

“The world changes quickly, and our applications need to adapt to these changes quickly. Program generation (as useful as it is) is but a link in a long chain” – Calton Pu.

Software evolution is inevitable, and evolution is part of maintenance. Maintenance is the most expensive part of a program’s life cycle. To minimize costs, we ideally would like to automate as many maintenance tasks as possible.

Cognitive systems is an exciting area of contemporary research [8]. A *cognitive system* is a program whose performance improves as it gains knowledge and experience. In effect, it is a program that automates some of its maintenance tasks. So how have relational optimizers fared?

It turns out that relational optimizers are cognitive systems! Query optimization relies on cost models that are driven by database statistics. Example statistics include the number of tuples in a relation, the selectivity of predicates (e.g., what fraction of a relation’s tuples satisfy a given predicate), the length of attribute values, etc. [22]. These statistics change over time as data is inserted and removed from tables. Thus, keeping statistics up-to-date is a key problem for optimizer maintenance. Interestingly, most optimizers – even simple ones – refresh their database statistics automatically. As generated query evaluation programs execute, statistics are gathered on data that is retrieved, and are used to refresh or update previously stored statistics. This allows the optimizer to improve the programs that it subsequently generates. In this way, optimizers learn new behaviors automatically and thus are cognitive systems.

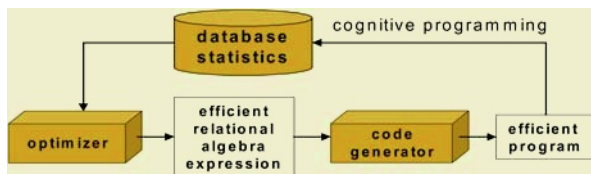


Fig. 2. Cognitive Programming in Relational Query

So if this also can be replicated in other domains, would this be Utopia? No, but it is on the road...

### 2.3 Generating Non-code Artifacts

“How do we cope with non-binary non-code artifacts, e.g., manuals, tutorials, help files etc. that are specific for a generated product?” – Ulrich Eisenacker

Generating non-code artifacts is an important problem, for no other reason than generating source code for a target program is always insufficient. Architects routinely use multiple representations of programs to express a system’s design, using representations such as process models, UML models, makefiles, and documents [13]. The productivity gains of generators that automatically synthesize source code is negated if other representations (which might be needed for integrating this program into a larger system) must be written manually. Clearly, *all* needed representations of a program should be generated. What are the experiences of database researchers in this regard?

Interestingly, relational optimizers use multiple representations of programs. In fact, two different representations are maintained for each relational algebra operator. One is the source code representation of the operator, and the second is a cost model representation, which is needed for optimization. Thus, for a relational algebra expression  $E$ :

$$E = \text{join}(\text{select}(\dots), \text{select}(\dots))$$

an optimizer derives a cost model of that program by composing cost model representations of the operators:

$$E_{\text{cost}} = \text{join}_{\text{cost}}(\text{select}_{\text{cost}}(\dots), \text{select}_{\text{cost}}(\dots))$$

In a similar manner, an optimizer can also derive source code representation of an expression by composing the code representations of these same operators:

$$E_{\text{code}} = \text{join}_{\text{code}}(\text{select}_{\text{code}}(\dots), \text{select}_{\text{code}}(\dots))$$

*That is, the modularity and structure imposed on code (or code generators) is exactly the same as that for cost models: they all align along operator boundaries.*

During optimization, optimizers synthesize cost models for each program they evaluate. They search the space of equivalent programs, synthesize the cost model of each program, use the cost model to estimate the performance of that program, and identify the program that will be the most efficient. Code is synthesized only for the most efficient program.

So the need for multiple representations is indeed present, but the approach is not sufficiently developed for general purpose needs. Recent work suggests it is indeed possible to synthesize arbitrary representations of programs using an algebraic approach [5].

### 2.4 The Quality of Performance (or Cost Model) Estimates

“A performance estimate of good quality, especially for modern hierarchical parallel systems, is needed.” – Holger Bischof

“Especially important are rules and methods for composing skeletons in large-scale applications with reliable performance prediction.” – Sergei Gorlatch

The quality of performance estimates has long been known to be critical for identifying good access plans in query optimization. Interestingly, cost estimates used by query optimizers have historically been simple and crude, based on averages. For  $n$ -way joins (for large  $n$ ) estimates are known to be very poor [15]. Further, performance estimates that are based on page caching – that is, knowing what pages are on disk and which are in the cache, are highly unreliable. Despite these limited capabilities, relational optimizers have done quite well. I am sure there are domains other than query processing that require more precise estimates.

In any case, if these problems are solved, would this be Utopia? No, its on the road...

## 2.5 Extensibility of Domain Algebras

“A key technical problem is expressiveness: can we fix a general purpose set of skeletons (read: operators) that covers all the interesting structures? Should we have several different skeleton sets for different application domains?” –

Susanna Pelagatti

“Expandability of skeleton frameworks must be studied. The programming model of skeletons must provide open source skeleton libraries/frameworks.”

– Marco Danelutto

“DSLs are best understood in terms of their ‘negative space’ – what they don’t do is just as important as what they do... How to avoid ‘mission creep’ for languages?” – Shriram Krishnamurthi

Are domain algebras closed, meaning do they have a fixed set of operators, or are domain algebras open, allowing new operators to be added subsequently? This is a fundamental question whose answer is not immediately obvious. The experience of database researchers is quite interesting with respect to this topic. To appreciate the lessons that they learned (and the lessons that we should learn), it is instructive to see what database researchers did “right”.

The success of relational query optimization hinged on the creation of a science to specify and optimize query evaluation programs. Specifically researchers:

- identified the fundamental operators of this domain, which was relational algebra,
- represented programs as equations (or expressions) which were compositions of these operators, and
- defined algebraic relationships among these operators to optimize equations.

Compositional programming is a holy grail for programming paradigms: it defines a set of building-blocks or “legos” that can be snapped together to build



different programs. The key property of *compositionality* is made possible by algebraic models. Compositionality is the hallmark of great engineering, of which relational query optimization is an example.

Now, let's return to the open or closed nature of domain algebras. Relational algebra was originally defined by the project, select, join, and cross-product operators. For years, *by definition* it was closed<sup>1</sup>. During this time, people were trying to understand the implications of the classic 1979 Selinger paper on System R optimizer [22], which revolutionized query evaluation and set the standard relational optimizers for the next two decades. This paper dealt only with queries formed from compositions of the basic relational operators. But from 1985 onward, there was a series of papers approximately titled "I found yet another useful operator to add to relational algebra". Among these operators are data cube (aggregation) [10], transitive closure [16], parallelization of query evaluation programs (that map a sequential program to a parallel program) [9], and new operators for time series [24], just to mention a few.

So is relational algebra complete? No! It is obvious now that it will never be closed, and will never be complete. There will always be something more. And this will be true for most domains. Database systems now deal with open algebras, where new operators are added as needed; they are more type extensible than before; hooks are provided into the optimizer to account for the peculiarities of new operators, such as Hellerstein's work [14] on user-defined queries.

But the core of model remains fixed: query optimization is still based on an algebra and programs are still represented algebraically, because the algebraic paradigm is simply too powerful to abandon.

## 2.6 Implications with Open Algebras

"Once a synthesis system solves non-trivial problems, it usually gets lost in vast search spaces which is not only spanned by the different ways to derive a specific program, but also the set of all possible programs satisfying the specification. Control of the search is thus a major problem, specifically the comparison and choice between different 'equivalent' programs." –

Bernd Fischer

"How to represent the space of possible optimization alternatives for a component (read: operator), so that the best combination of optimizations can be chosen when the component is used?" – Paul Kelly

Given the fact that open algebras will be common, how has this impacted query processing research? My guess is that the database community was lucky. The original optimizers supported only the initial set of relational operators. This constraint made query optimization amenable to a dynamic programming solution that admitted reasonable heuristics [22]. The end result was that database

---

<sup>1</sup> Not "closed" in a mathematical sense, such as addition is closed in integers but not in subranges. By "closed" I mean a social club: no new members were thought to be needed.

people could legitimately claim that their optimization algorithms were guaranteed to find the “best” query evaluation program. And it was this guarantee that was absolutely crucial for early acceptance. Prior work on query optimization used only heuristics, and the results were both unsatisfying and unconvincing. Providing hard guarantees made all the difference in the world to the acceptance of relational optimizers.

Ironically, the most advanced databases today use rule-based optimizers that offer many fewer guarantees. But by now, database people are willing to live with this. So is this Utopia? No, its on the road. . .

## 2.7 What Is Domain Analysis?

“A key problem is what exactly are the common algorithmic structures which underpin ‘enough’ parallel algorithms to be interesting? Is it reasonable to expect that such a generic collection exists, or is it more appropriate to look in domain specific ways?” – Murray Cole

“What is the target domain?” is a core question of generator technologies. An analysis of a domain, called *domain analysis*, identifies the building blocks of programs. Generators implement the output of domain analysis. But what exactly is “domain analysis” and what should be its output? Today, “domain analysis” is almost a meaningless term. But oddly enough, whatever it is, we all agree domain analysis is important! For instance:

“On domain engineering – any synthesis system which is useful to users must be able to generate a large number of non-trivial programs which implies it must cover a substantial part of the domain of interest. Formalizing and organizing this domain is the major effort in building a synthesis system.” – Bernd Fischer

“We need a methodology to systematically map domain analysis into a DSL design and implementation.” – Charles Consel

“We need a systematic way to design DSLs.” – Krzysztof Czarnecki

“Future work should improve existing methodologies for DSLs” –  
Laurent Reveillere

So what did database people do? They had two different outputs of domain analysis. First, they defined relational algebra, which is the set of operators whose compositions defined the domain of query evaluation programs. (So defining a domain algebra is equivalent to defining the domain of programs to generate). Another related analysis produced the SQL language, which defined declarative specifications of data retrieval that hid its relational algebra underpinnings. So database people created both and integrated both.

In general, however, these are separable tasks. You can define a DSL and map it to a program directly, introducing optimizations along the way. Or you can define a DSL and map it to an algebra whose expressions you can optimize.

This brings up a fundamental result on hierarchies of DSLs and optimizations. The first time I saw this result was in Jim Neighbor’s 1984 thesis on

DRACO [19]. The idea is simple: programs are written in DSLs. You can transform (map) an unoptimized DSL program to an optimized DSL program because the domain abstractions are still visible. Stated another way, *you can not optimize abstractions that have been compiled away*.

Given an optimized DSL program, you translate it to an unoptimized program in a lower level abstraction DSL and repeat the same process until you get to machine code. So it is this “zig-zag” series of translations that characterize hierarchies of DSLs (or hierarchies of languages, in general) and their optimizations (Figure 3a).

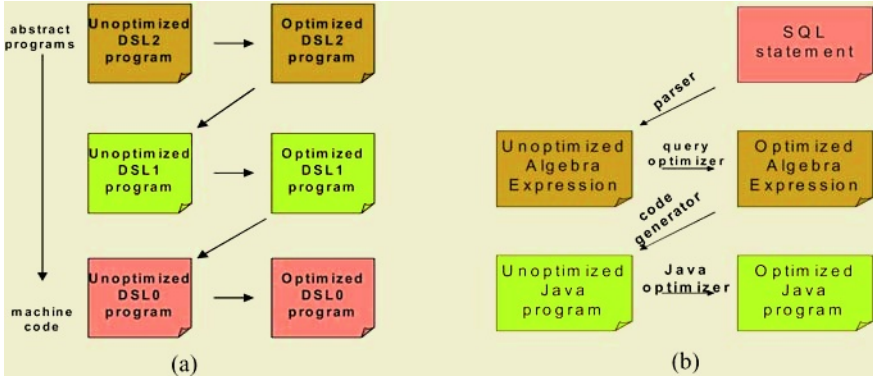


Fig. 3. DRACO DSL Hierarchies and Optimizations

Figure 3b depicts a DRACO view of relational query optimization. You begin with an SQL SELECT statement. A parser produces an unoptimized relational algebra expression. A query optimizer optimizes this expression and produces an optimized relational algebra expression. A code generator translates this to an unoptimized Java program, and the Java compiler applies its optimization techniques to produce an optimized Java program.

GP occurs when mapping between levels of abstraction, and AP occurs when optimizing within a level of abstraction. More commonly, optimizations are done internally by DSL compilers. That is, a DSL program is mapped to an unoptimized internal representation, and then this representation is optimized before translating to an executable.

The point is that there can be different outputs of domain analysis, and that different optimizations occur between various translations. The most significant optimizations, I assert, occur at the “architectural” or “algebraic” level.

So is this Utopia? No, its on the road...

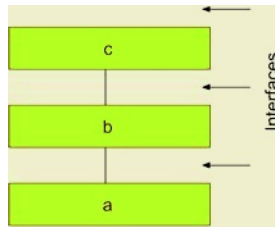
## 2.8 Software Design

“Software technology in domain-specific programming involves getting the interface right, getting the split right (how to separate the domain-specific from the domain-independent)” – Chris Lengauer

“It is most essential for component software to standardize well-defined interfaces.” – Wolfgang Weck

These are fundamental problems of software design. I’ll sketch a common problem, and then show how a database approach solves it. But first, we need to understand the relationship between operator compositions and layered designs.

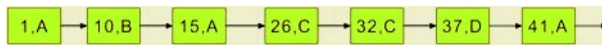
Figure 4 depicts a layered design, where layer **a** is on the bottom, layer **b** sits atop **a**, and layer **c** is atop **b**. Operator implementations often correspond to layers or layers of abstraction. The design in Figure 4 corresponds to the composition  $c(b(a))$ , where layers **a**, **b**, and **c** implement their respective operators.



**Fig. 4.** Layering Interpretation of Composition  $c(b(a))$

In general, systems are conceptually, but *not* physically, layered [12]. Interfaces delineate the boundaries of operators/layers **a**, **b**, and **c**. These interfaces might be Java interfaces or *they might be DSL specifications!*

Now to an example. Suppose a program maintains a set of records of form (**age**, **tag**) and these records are stored on an ascending age-ordered linked list (Figure 5). Here the first record has **age**=1, **tag**=A, the next record **age**=10, **tag**=B and so on.



**Fig. 5.** A Linked List for our Example

Periodically, we want to count all records that satisfy the predicate **age**>**n** and **tag**=**t**, for some **n**, **t**. What is the code for this retrieval? Here’s our first try: we write an ordered list data type. Our retrieval is simple: we examine every record, and apply the full predicate. This program is easy to write. Unfortunately it is inefficient.

```
int count = 0;
Node node = container.first;
while (node != null) {
    if (node.tag == t && node.age > n)
        count++;
    node = node.next;
}
```

Our next try exploits a property of ordered lists. The observation is that we can skip over records that don't satisfy the key predicate, `age>n`. As soon as we find the first record that satisfies the predicate, we know that all records past this point also satisfy the predicate, so all we need to do is to apply the residual predicate, `tag==t`, to the remaining records. This leads to the following 2-loop program. It takes longer to write, longer to debug, but the result is more efficient.

```
int count = 0;
Node node = container.first;

while (node != null && node.age <= n)
    node = node.next;

while (node != null) {
    if (node.tag == t)

        count++;
    node = node.next;
}
```

There is yet another alternative: a Java programmer would ask: Why not use the Java library? With library classes, you would have to write even less code! In the J2SDK, `TreeSet` implements `SortedCollection`. With `TreeSet`, the desired subcollection can be extracted in one operation (called `tail()`). By iterating over the extracted elements and applying the residual predicate as before we can produce the desired result. The code is indeed shorter:

```
int count = 0;
// ts is TreeSet with elements

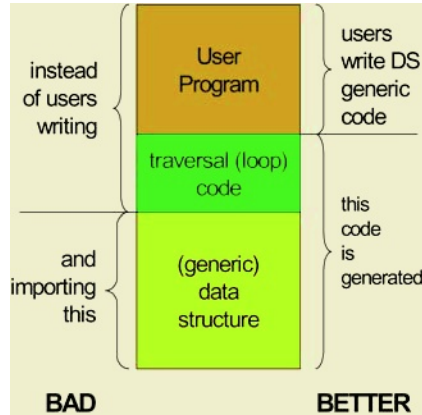
TreeSet onlyOld = ts.tail(n);
Iterator i = onlyOld.iterator();
while (i.hasNext()) {
    Node node = (Node) i.next();
    if (node.tag == t)
        count++;
}
```

Unfortunately, the `TreeSet` code is *much* slower (maybe even slower than original list implementation). Why? The reason is `TreeSet` creates an index over all the records that it sorts. Applying the `tail` operation creates an index that references the extracted elements. Index construction can be very slow. This raises a classical dilemma: if you want execution speed, stay with customized code. If you want to write the code fast, use libraries.

This problem has everything to do with selecting the right interfaces and right abstractions and is a classical situation for the use of DSLs and GP. Let

me review a database solution to this problem [3]; a more general formulation is presented in [21].

Figure 6 depicts the code we wrote. Users would write the code above the left horizontal line, which is the user program and traversal loop. But look what happens when we swap data structures (or change the record ordering). Our program breaks. The reason is that implementation details – specifically the record storage order – of the data structure leaked into our program. So if these details change, our program has to change too.



**Fig. 6.** What To Generate

What we need are higher-level abstractions (e.g., a DSL) to specify a data structure and its traversals. That means that users should write data structure generic code, and the rest (loop, data structure itself) is generated. (That is, users should write the code above the right horizontal line in Figure 6). A database solution is to use a SQL to specify retrieval predicates and declarative relation implementation specifications to implement the container. That is, a container is a relation. Predicates for cursors (iterators) are declaratively specified in SQL and the SQL compiler generates the cursor/iterator class that is specific to this retrieval. This makes the user program data-structure (relation implementation) independent! Further, it exploits DSLs, algebraic optimizations of relational optimizers and amazingly, the code a user has to write is even simpler than using generics! And the code is efficient [4], too!

```
cursor1 c = new cursor1(n,t);
int count = 0;

for (c.first(); c.more(); c.next())
    count++;
```

So is this Utopia? No, its on the road...

## 2.9 Scalability of Algebraic Approaches

Do algebraic approaches scale? Let’s face it, query evaluation programs and data structures are tiny. Can an algebraic approach synthesize large systems? If it can’t, then we should pack our bags and try something else.

Interestingly, it does scale. And this brings us to a key result about scaling. *Feature Oriented Domain Analysis (FODA)* is pioneering work by Kyo Kang, et al [17]. His work deals with product-lines and producing a family of related applications by composing primitives.

So the goal is to synthesize a large application from “primitives”. But what are these primitives? Consider the following thought experiment. Suppose you have programs that you want others to use. How would you describe them? Well, you shouldn’t say what DLLs or object-oriented classes each uses. No one will care. Instead, you are more likely to describe the program by the features it has. (A *feature* is a characteristic that is useful in distinguishing programs within a family of related programs [11]). For example, you might say **Program1** has features **X**, **Y**, and **Z**. But **Program2** is better because it has features **X**, **Q**, and **R**. The reason is that clients have an understanding of their requirements and can see how features relate to requirements.

A common way to specify products is by its set of features. While this is almost unheard of in software, it is indeed common in many other engineering disciplines. For example, go to the Dell Web site. You’ll find lots of web pages that provide declarative DSL specifications (e.g. menu lists) from which you can specify the features that you want on your customized PC. After completing this specification, you can initiate its order. We want to do the same for software.

Here is a program synthesis vision that has evolved concurrently with FODA [2]. Program **P** is a package of classes (**class1-class4**). **P** will have an algebraic definition as a composition of features (or rather feature *operators*). Consider Figure 7. **P** starts with **featureX**, which encapsulates fragments of **class1-class3**. **featureY** is added, which extends **class1-class3** and introduces **class4**, and **featureZ** extends all four classes. Thus, by composing features which encapsulate fragments of classes, a package of fully formed classes is synthesized.

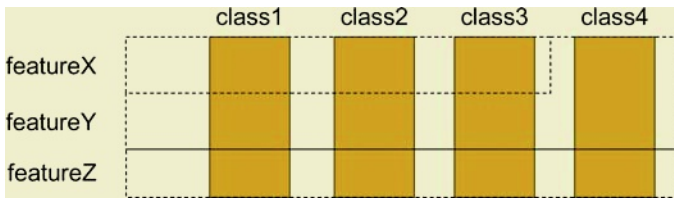


Fig. 7. Program **P** = **featureZ**( **featureY**( **featureX** ) )

This algebraic approach is related to the age-old concept of *step-wise refinement* which asserts that a complex program can be constructed from a simple

program by adding details (in this case, features), one at a time. This means that feature operators are implemented by program refinements or program extensions<sup>2</sup>. Program P is created by starting with a simple program, **featureX**. This program is extended by **featureY** and then by **featureZ** – a classic example of step-wise development.

Here is an example problem that illustrates the scalability of algebraic approaches. I and my students are now building customized tools for processing programs written in extensible- Java languages. These tools belong to *Integrated Development Environments (IDEs)*. The GUI shown in Figure 8 is a declarative DSL. We allow architects to select the set of optional Java extensions that they want (in the left-most panel), the set of optional tools that they want (in the middle panel), and by pressing the Generate button, the selected tools will be synthesized and will work for that specified dialect of Java. We are now generating over 200K Java LOC from such specifications, all of which are (internally) driven by equations. So algebraic approaches do indeed scale, and there seems to be no limit to the size of a system that can be produced [6].

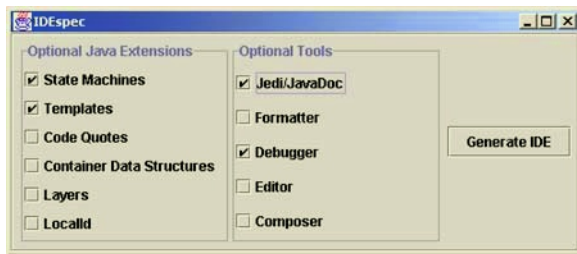


Fig. 8. A Declarative GUI Interface for Customized IDE

So is this Utopia? No, its on the road. . .

## 2.10 Tool Support

“One of the biggest technical problems is that metaprogramming and generative programming are not directly supported by programming languages. A cleaner and safer mechanism (other than (ab)using the template mechanism and type system in C++) is clearly needed.” – Andrew Lumsdaire

<sup>2</sup> The term “refinement” is often used in the context of adding implementation details *without* modifying the original specification. “Extensions” add details to implementations and can enhance the original specification. The definition of these terms is not agreed upon in the OO and programming methodology communities, and my informal use of these terms reflects this lack of consistency. Still, it remains an open problem to relate notions of program synthesis using features to concepts in algebraic program specifications. Defining these relationships clearly should lead to a valuable advance in both areas: a formal theory for practical generative programming, and a practical outlet for theory.



“The ‘big’ technical challenge is getting a good infrastructure for developing generators that includes extensible programming languages transformation systems, and metaprogramming libraries.” – Yannis Smaragdakis

“Type safe program generation and efficient implementation of multi-stage languages are important.” – Walid Taha

Long ago, I realized that the scale of programs will have an impact on the tools that are used. Some domains will have operators that can be implemented by comparatively simple macro expanding techniques. Figure 9 shows a pair of classes that implement a bare-bones singly-linked list program shown in black, non-italic font. By applying an doubly-linked list operator, this program is transformed into a program that is a doublylinked list. The transformation-added code is shown in *red, italic* font. Relatively simple tools can be built to achieve this kind of program rewriting.

```
class list {
    Node first;
    Node last;

    void insert( Node x ) {
        if (last == null)
            last = x;
        x.next = first;
        first = x;
        x.prior = null;
    }
}

class Node {
    Node next;
    String value;
    Node prior;
}
```

Fig. 9. Transformation of a Singly-Linked List into a Doubly-

Surprisingly, this simple approach has worked well for *large* applications. However, for smaller programs or algorithm synthesis, much more is needed. The work at Kestrel on synthesizing scheduling algorithms [7] and the synthesis and optimization of orbital algorithms at NASA Ames [23] are really impressive. They require nontrivial “domain theories” and a non-trivial programming infrastructure. Even the simple data structures domain requires more sophistication than macros.

One reason for this is that domain-specific optimizations are below the level of relational algebraic rewrites; one has to break encapsulation of abstractions to achieve better performance. And the operations (functions) seem much more complicated.

Oddly, the synthesis of large systems has different requirements. Architects generally don’t care about low-level optimization issues. The problem is more of gluing operators together; breaking encapsulations to optimize is rarely done. For years I thought `lispquote-unquote` features were critical for all generative tools. I now think that most domains don’t need such sophistication. Tools for synthesis-in-the-large will be very different than those for synthesis-in-the-small.

There seems, however, to be a pleasing result: there is one way to conceptualize program families using features, but how operators are implemented is domain-dependent. There are lots of ways to implement operators: as macros, lisp-quote-unquote, program transformation systems, objects, etc. It is a matter of choosing the right implementation technology. However, the process by which one identifies the fundamental operators in a domain is largely independent of operator implementation.

So is this Utopia? No, its on the road. . .

## 2.11 Verification

“What kinds of verifications are appropriate/feasible for what kinds of languages, and what approaches are appropriate to carry out these verifications? How can languages be designed to facilitate verification?” – Julia Lawall

We want guarantees about generated programs. We want proofs that properties of algorithms that implement operators are not violated when operators are composed. What are the lessons learned by database researchers?

As far as I can tell, verification has never been an issue. And it is not surprising either. Historically there are no (or trivial) synchronization issues, no real-time issues in query evaluation programs. Query evaluation programs in database systems wrestle with scale and performance, not correctness issues.

I want to point out that there is important work on verifying programs using features. Most properties that are to be preserved in compositions are local properties of features. You want to prove that feature properties are not violated by composition. I recommend reading the Li, Krishnamurthi, and Fisler paper [18] to get a flavor of this line of work.

So is this Utopia? No, its on the road. . .

## 2.12 Technology Transfer

“How do we make our concepts accessible to ‘Joe Parallel Programmer’, who knows Fortran/C+MPI/Threads and is no so unhappy with these?” –

Murray Cole

“We need to reduce the necessary expertise to use generative programming and metaprogramming tools for DSL definition and implementation. The use of current systems is very much a craft.” – David Wile

Technology transfer are tough issues indeed. By far, technology transfer is the hardest problem. Education is the key. We must demonstrate over and over again where GP, DSLs, and AP are relevant and beneficial. We must be constantly looking for new applications to demonstrate their value. Sadly, I fear, not until large companies like Microsoft see the advantage, progress will be glacial. You have heard of the 17 year lag between the discovery of ideas and practice; I think things are much longer for software engineering simply because the inertia is so great.

So is this Utopia? No, its on the road. . .

### 2.13 And More!

“What are the relative merits of different programming models?” –

Prem Devanbu

“Language design and hiding the meta-level is an important problem.” –

Joerg Striegnitz

“The economic issues (cost, time-to-market, maintenance, flexibility) are not well understood.” – Chris Ramming

There is no lack of other issues. Every issue raised above is indeed important. Often the progress of a field hinges on economics. And until we understand the economic ramifications (i.e., benefits), transfer of our ideas to industry will be slow.

## 3 Epilog

So if we solved all of the previously mentioned problems, would this be Utopia? It might be. But let’s put this in perspective: Did database people know they were on the road to Utopia? Hardly. Let’s start with Codd’s 1970 seminal paper on the Relational Model. Its first public review in *Computing Surveys* panned the idea [20]. And it is easy to forget that the appreciation of the Relational Model grew over time.

“It isn’t like someone sat down in the early 1980’s to do domain analysis. No – we had trial and error as briefly outlined:

- (1) CODASYL 1960s – every update type and every query type requires a custom program to be written,
- (2) Codd 1970 – relational algebra – no keys but in theory no custom programs,
- (3) IBM & researchers (late) 1970s – compelling business issues press development at business labs and universities. Query languages, schema languages, normal forms, keys, etc.
- (4) Oracle early 1980s – and they are off...

Now, which domain analysis methodology shall we assert could have achieved this in a shorter term? It takes time and experience on the road to a solution; it also takes the patience to continually abstract from the problem at hand until you recognize you already have a solution to the immediate problem.” –

Jim Neighbors

In short, it takes time and clarity of hindsight to find Utopia. Utopia is a small place and is easy to miss.

“People of different backgrounds have very different opinions on fundamental problems and principles of software engineering, amazingly.” – Stan Jarzabek

“I am not convinced that any of these problems is concretely enough defined that if solved I will recognize the solution as a big advance. I believe the area is in a state where there is no agreement as to what will constitute the next big step.” – Yannis Smaragdakis

“Mindset is a very important issue. How can researchers find Utopia if they are not trying to get there? How can they be trying to get there if they are not solving a specific problem? Without a problem, they are on the road to where?” – Jim Neighbors

My response: this is Science. The signs along the road to scientific advancement are strange, if not obscure. But what did you expect? Some observations and results will be difficult, if not impossible to explain. But eventually they will all make sense. However, if you don’t look, you’ll just drive right past Utopia, never knowing what you missed.

My parting message is simple: database researchers got it right; they understood the significance of generative programming, domain-specific languages, automatic programming and lots of other concepts and their relationships, and they made it all work.

Software engineering is about the challenges of designing and building large-scale programs. The future of software engineering will require making programs first-class objects and using algebras and operators to manipulate these programs. Until these ideas are in place, we are unlikely to reach Utopia. Our challenge is to replicate the success of database researchers in other domains. I believe that our respective communities – generative programming, metaprogramming, and the skeleton communities – represent the future of what software engineering will become, not what it is today.

I hope to see you on the road!

## Acknowledgements

I am grateful to Chris Lengauer and Jim Neighbors for their comments and insights on an earlier draft of this paper.

## References

1. R. Balzer, “A Fifteen-Year Perspective on Automatic Programming”, *IEEE Transactions on Software Engineering*, November 1985.
2. D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM TOSEM*, October 1992.
3. D. Batory, V. Singhal, J. Thomas, and M. Sirkin, “Scalable Software Libraries”, *ACM SIGSOFT* 1993.
4. D. Batory, G. Chen, E. Robertson, and T. Wang, “Design Wizards and Visual Programming Environments for GenVoca Generators”, *IEEE Transactions on Software Engineering*, May 2000, 441-452.
5. D. Batory, J.N. Sarvela, and A. Rauschmayer, “Scaling Step-Wise Refinement”, *International Conference on Software Engineering (ICSE-2003)*.
6. D. Batory, R. Lopez-Herrejon, J.P. Martin, “Generating Product-Lines of Product-Families”, *Automated Software Engineering Conference*, 2002.
7. L. Blaine, et al., “Planware: Domain-Specific Synthesis of High-performance Schedulers”, *Automated Software Engineering Conference 1998*, 270-280.

8. R.J. Brachman, "Systems That Know What They're Doing", *IEEE Intelligent Systems*, Vol. 17#6, 67-71 (Nov. 2002).
9. D. DeWitt, et al., The Gamma Database Machine Project, *IEEE Transactions on Data and Knowledge Engineering*, March 1990.
10. J. Gray, et al. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", *Data Mining and Knowledge Discovery* 1, 29-53, 1997.
11. M. Griss, "Implementing Product-Line Features by Composing Component Aspects", *Software Product-Line Conference*, Denver, August 2000.
12. A.N. Habermann, L. Flon, and L. Coopridge, "Modularization and Hierarchy in a Family of Operating Systems", *CACM*, 19 #5, May 1976.
13. A. Hein, M. Schlick, R. Vinga-Martins, "Applying Feature Models in Industrial Settings", *Software Product Line Conference (SPLC1)*, August 2000.
14. J. Hellerstein, "Predicate Migration: Optimizing Queries with Expensive Predicates", *SIGMOD* 1993.
15. Y.E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results", *ACM SIGMOD* 1991.
16. Y. E. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators", *Very Large Database Conference 1986*, 403-411.
17. K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study". Tech. Rep. CMU/SEI-90-TR-21, Soft. Eng. Institute, Carnegie Mellon Univ., Pittsburgh, PA, Nov. 1990.
18. H.C. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for Modular Feature Verification", *Automated Software Engineering Conference 2002*, 195-204.
19. J. Neighbors, "Software construction using components". Ph. D. Thesis, (Technical Report TR-160), University of California, Irvine, 1980.
20. J.E. Sammet and R.W. Rector, "In Recognition of the 25th Anniversary of Computing Reviews: Selected Reviews 1960-1984", *Communications of the ACM*, January 1985, 53-68.
21. U.P. Schultz, J.L. Lawall, and C. Consel, "Specialization Patterns", Research Report #3835, January 1999.
22. P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database System", *ACM SIGMOD* 1979, 23-34.
23. M. Stickel, et al., "Deductive Composition of Astronomical Software from Subroutine Libraries", In *Automated Deduction*, A. Bundy, ed., Springer-Verlag Lecture Notes in Computer Science, Vol. 814.
24. M. Stonebraker, "Inclusion of New Types in Relational Data Base Systems", *IEEE Data Engineering Conference*, Los Angeles, CA, February 1986.

# From a Program Family to a Domain-Specific Language

Charles Consel

INRIA/LaBRI

ENSEIRB – 1, avenue du docteur Albert Schweitzer,

Domaine universitaire – BP 99

33402 Talence Cedex, France

`consel@labri.fr`

`http://compose.labri.fr`

**Abstract.** An increasing number of domain-specific languages (DSLs) are being developed and successfully used in a variety of areas including networking, telecommunications and financial products. Yet, the development of a DSL is still an obscure process and its assessment is often partial.

This paper proposes to structure the development of a DSL on the notion of program family. We outline the main steps of such development. Furthermore, we argue that a program family provides a basis to assess a DSL.

The ideas discussed in this paper are directly based on our experience in developing DSLs for various domains and studying existing ones. We illustrate these ideas with various examples of DSLs.

## 1 Introduction

Domain-specific languages (DSLs) are being successfully designed, implemented and used, both academically and industrially, in a variety of areas including interactive voice menus [1], Web services [2], component configuration [3], financial products [4], and communication services [5]. Yet, developing a DSL is still a complicated process, often conducted in an ad hoc fashion, without much assessment of the resulting language.

A step toward a methodology for DSL development is presented by Consel and Marlet [6]. It is based on a denotational semantic approach and integrates software architecture concerns. The development of a DSL is initiated with a variety of ingredients, mainly consisting of a program family, technical literature and documentation, knowledge of domain experts, and current and future requirements. From these ingredients, domain operations and objects (*i.e.*, values) are determined, as well as the language requirements and notations. Notations are further refined to produce a DSL syntax. Its semantics is informally defined: it relates syntactic constructs to the objects and operations identified previously. Next, like any programming language, the DSL semantics is divided into two stages: the static semantics, that can be viewed as program configuration

computations, and the dynamic semantics, that corresponds to the computations of a configured program. The staged semantics is then formalized in the denotational setting. The static semantics is specified as interpretation functions. The dynamic semantics forms a dedicated abstract machine that defines the execution model of the DSL.

In this paper, we revisit this methodology from a pragmatic viewpoint in the light of various DSLs that we have developed since then. Following the work of Weiss [7], these activities illustrated the key role of a program family in developing a DSL. Using the program family as the main thread, we outline the basic steps of DSL development. First, we observe that a program family naturally gives rise to the development of a library of functions (in the case of a procedural language). Such library is a natural starting point to design a dedicated abstract machine. Indeed, library entries already correspond to fundamental operations in the domain. They can be used as a basis to define domain-specific instructions. Furthermore, library functions often manipulate possibly scattered pieces of data that can contribute to introduce a notion of machine state. Instructions and state are used to define a domain-specific abstract machine. This machine and common program patterns occurring in the program family serve as basic ingredients to develop a DSL. Once developed, we argue that a program family can also be used to assess a DSL.

Our discussion is illustrated by a DSL for device drivers, named Devil [8–10]. This DSL targets a family of programs (in fact, a family of layers) that supports the communications between the upper part of a device driver and a device. This communication layer often relies on a form of library of operations to manipulate bit-level values in the device and to update state elements. We discuss how these components can form an abstract machine dedicated to device communications. This abstract machine can, in turn, serve as an input to define a DSL aimed to combine abstract machine operations. Devil programs are written by device driver developers. Their compilation produces the implementation of a communication layer to be assembled with the upper part of the device driver.

To widen the scope of this paper, we also illustrate our discussion with the example of the Sun Remote Procedure Call (RPC) mechanism [11]. This mechanism is used to develop applications distributed over a network of possibly heterogeneous machines. A procedure can be invoked remotely; its arguments and returned value are automatically marshaled into and unmarshaled from a machine-independent format. In this paper, we concentrate on this (un-)marshaling layer, called XDR. For convenience, thereafter, the term “marshaling” is used instead of “(un-)marshaling”. This layer consists of a DSL and a compiler producing code gluing calls to a generic library. The programmer of a distributed application writes an XDR description that defines the type of the remote procedure arguments and return value. This description is compiled into the marshaling layer on both the client and server sides. This layer is invoked by the upper part of the distributed application.

The underlying program family of the XDR language is very different from device drivers, and thus, makes it a complementary example to Devil.

## Overview

Section 2 defines the notion of a program family and discusses its importance in the context of DSL development. Section 3 explains how a program family naturally gives rise to a library. Section 4 shows how a library can be used as a basis to develop a domain-specific abstract machine. Section 5 explains how to introduce a DSL from an abstract machine. Section 6 discusses the importance of a program family to assess a DSL.

## 2 Program Family

A program family is a set of programs that share enough characteristics that it is worthwhile to study them as a whole [12]. This set of related programs naturally share commonalities, that is, features, assumptions and computations that hold for all family members. These commonalities may exhibit some well-identified variations. Analyzing a program family is part of an activity called domain analysis, which itself is part of Domain Engineering. In short, Domain Engineering aims to collect, organize and store information about past software developments, and to exploit it for new developments. A step toward bridging Domain Engineering and DSL development is presented by Czarnecki and Eisenecker [3]. Our approach is more restricted: it aims to use programming language technology to identify and describe the systematic steps in designing a DSL from a library. Thus, various aspects covered by Domain Engineering are not considered by our work (*e.g.*, risk analysis and collection of knowledge from experts).

### 2.1 The Scope of Computations

Often, identifying a program family is an intuitive process. Device drivers represent a typical example of a program family: there exists a large set of programs devoted to operate different kinds of devices (*e.g.*, Ethernet, video, sound, disk, and DMA). Even a specific model of a specific device corresponds to many drivers to account for incremental hardware improvements and added features. A study of device drivers shows that they share a lot of commonalities, as reported by Réveillère *et al.* [9]. In general, such an observation is necessary to motivate the development of a DSL but it needs to be completed by a study of the *scope* of computations involved in the identified program family. Indeed, if the target program family consists of computations that are too general purpose, it may defeat some of the benefits of DSLs, like domain-specific verifications.

Let us illustrate this issue by examining our working example. Roughly speaking a device driver may be divided into three parts. The upper part is a communication layer with the operating system. It manages kernel resources in support for device operations, and exchanges information with user applications through the kernel. This top part consists of computations that are specific to an operating system and a programming language to a large extent. The middle part of a



device driver corresponds to its logic: it consists of arbitrary computations ranging from calculating a disk block number to managing an ethernet packet queue. The bottom part of a device driver, mentioned earlier, is devoted to communicating with the device at a very low level. It involves assembly-level operations to extract and build bit fragments and explicit I/O bus accesses.

## 2.2 Delimiting a Scope of Computations

As can be noticed, a close examination of the program family of device drivers reveals layers consisting of very different kinds of computations. Modeling the domain of device drivers with a single DSL would necessarily lead to a general-purpose language. To address this problem, one strategy consists of targeting a specific class of devices to reduce the scope of computations to be expressed. This strategy was used by Thibault *et al.*; they developed a single DSL, named GAL, to specifically address the domain of graphic display adaptors for the X11 server [13].

Another strategy to reduce the scope of computations is to develop a DSL for a specific layer. This strategy was used to develop Devil; this DSL specifically targets the interaction layer between the driver logic and the device.

A similar strategy was used to develop the XDR language that focuses on the marshaling layer. This DSL aims to declare a data structure to be marshaled in a machine-independent format.

Both Devil and XDR compilers generate an implementation of an interface, respectively, to communicate with the device, and to convert data.

## 3 From a Program Family to a Library

When a program family is identified, developers naturally exploit the situation by introducing re-usable software components. Abstracting over commonalities generally leads to the identification of domain-specific operations. Variabilities translate into versions of operations, variations of implementations, and/or parameterization. In practice, after some iterations, a program family leads to the development of a library. The objects manipulated by the library functions contribute to identify domain-specific objects.

In our working example, device drivers typically rely on a library of functions performing bit operations both common to most device drivers and dedicated to the device peculiarities. Other fundamental operations include reading from and writing to a device. Variabilities on such operations correspond to whether the device is accessed using memory operations because it is mapped in memory, or whether processor-dependent input/output operations should be used. Note that our notion of library refers to a collection of operations common to a given program family. This does not make any assumption on how the library is implemented. For example, in the case of device drivers, most library functions are either C macros or assembly language routines. This implementation strategy is motivated by stringent performance constraints. Also, it is a consequence of the very fine-grained nature of operations frequently occurring in a

device-communication layer (*e.g.*, bit-level operations). In fact, the narrow scope of computations involved in this layer does not require much parameterization in the library operations.

Other program families give rise to more generic libraries. As an example, consider the XDR library that consists of a marshaling function for each basic type. Each function is parameterized with respect to a variety of aspects such as the coding direction (marshaling/unmarshaling) and whether a value is coded from/to a stream or memory. These aspects are gathered in a data structure that is threaded through the marshaling code. Compared to the device driver example, the XDR library targets a program family consisting of a much larger scope of computations. Another key difference is that performance was not a critical requirement at the time the XDR layer was developed in the mid-eighties. This situation translated into highly-generic, coarse-grained library functions.

## 4 From a Library to an Abstract Machine

The abstract machine of a DSL defines the run-time model. It is dedicated to the target domain in that it is composed of a domain-specific set of instructions and a domain-specific state. A library is a key ingredient to the definition of a domain-specific abstract machine.

### 4.1 Domain-Specific Instructions

The development of a library for a program family exposes common operations that can be used as a basis to design instructions for a domain-specific abstract machine. In contrast with library functions that are directly used by a programmer, domain-specific instructions are assumed to be generated by a compiler or invoked by an interpreter. Consequently, re-use is not subject to the programmer's rigor; it is guaranteed by definition of the compiler or the interpreter. Since re-use is systematic, the design of domain-specific instructions can trade usability for other concerns. For instance, one can ease the development of a compiler by introducing coarse-grained, highly-parameterized instructions. Alternatively, one can develop highly-customized versions of the same operations to account for different optimization opportunities, and rely on the compiler to select the appropriate version.

A typical difference between parameterized and customized instructions is interpretation of instruction operands that mostly takes the form of checks. These checks can either be performed statically (at compile time) or dynamically (at run time). For instance, in the XDR example, a lot of buffer overflow checks can be performed statically. As a result, a marshaling instruction of a basic type could have two versions: one with the buffer overflow check and one without. Where and when each version should be used is an issue that already pertains to the DSL to be defined upon the abstract machine; this issue is discussed in Section 5.

## 4.2 Domain-Specific State

A library also contributes to identify some notion of state manipulated by domain-specific operations. In the device driver example, domain-specific operations manipulate an implicit state. The notion of state is in fact intrinsic to the device; transitions are caused by specific read and write operations on the device registers. For example, when a device offers several emulation modes, one such mode is selected by writing some value in one of the device registers. The state resulting from such write operation may not need to be represented with some data structure and threaded through the device driver, unless it ensures the safety of some instructions. For instance, an emulation mode may make some registers accessible. Thus, recording the emulation status as a part of the state may guarantee the correctness of register accessibility.

Unlike the device driver example, most library functions explicitly pass arguments representing some notion of state. For example, in the XDR library, a data structure is threaded through the marshaling code. This data structure can be viewed as some sort of marshaling state, composed of elements such as the coding direction (marshaling/unmarshaling), a pointer to the data being processed and the buffer space left for the marshaling. Here again, the state may be checked to ensure the safety of some operations, like avoiding a buffer overflow.

As can be seen, whether implicit or explicit, the state of a domain-specific abstract machine is an important component: not only to model the run-time environment but also to ensure the safety of some instructions. Identifying the elements of the machine state and studying their dependencies on machine instructions is a key step toward developing a domain-specific abstract machine. Yet, to complete this development the DSL also needs to be taken into account.

## 5 From an Abstract Machine to a DSL

A program family can guide the design of a DSL in different ways. It contributes to determine whether a language action is performed at compile time or at run time. It is used to extract common program patterns that can suggest syntactic constructs. Finally, it exhibits implementation expertise that can be turned into domain-specific optimizations.

### 5.1 Staging the DSL Actions

As illustrated by the buffer overflow check, defining the abstract machine of a DSL requires to determine whether an action is to be performed at compile time or postponed until run time. Traditionally, the compiler actions correspond to the static semantics of a language, whereas the run-time actions correspond to the dynamic semantics [14]. Deciding whether an action belongs to the static or the dynamic semantics has consequences on the language design. As a typical example, consider type checking. For a language to be statically typed requires specific constraints (*e.g.*, a unique type for conditional branches) and/or extra declarations (*e.g.*, a union type constructor).

Like a general-purpose language (GPL), the designer of a DSL needs to make choices regarding the stage of actions. In the context of the XDR example, one such choice regards the buffer overflow check. The XDR library includes marshaling functions for fixed and variable-length objects. The former objects do not require any check but the latter objects do. Thus, one approach consists of assuming the worst and systematically performing a buffer overflow check. This approach is illustrated by RPCGEN, the compiler for the XDR language. This greatly simplifies the compilation process: because each marshaling function encapsulates buffer overflow checks, in effect, the compiler delegates this language action to the run-time environment. The remaining compilation process mostly amounts to gluing marshaling functions together. Of course, this advantage comes at the expense of efficiency which was not the primary concern of the SUN RPC developers.

In the case of the device driver example, the most critical requirement of this domain is performance. Thus, determining the stage of the DSL actions should aim to improve the safety of device drivers written in the DSL compared to those written in a GPL, but without compromising performance. This issue is illustrated by register accessibility in the presence of emulation modes. We designed Devil so as to make it possible to statically check register accessibility. This static check has required the introduction of some compiler state to maintain the emulation status.

This topic is further studied by Consel and Marlet who detail and illustrate the staging of the formal definition of a DSL in a denotational setting [6].

## 5.2 Program Patterns

The design of a DSL can also take advantage of a specific analysis of its program family. This analysis is aimed to identify repetitive program patterns. These patterns can be abstracted over by introducing appropriate syntactic constructs.

For example, to optimize memory space, a device register often consists of a concatenation of values. As a consequence, device drivers exhibit repetitive sequences of code to either extract bit segments from a device register, or to concatenate bit segments to form a register value. To account for these manipulations, Devil includes concise syntax to designate and concatenate bit fragments.

In the XDR example, like most session-oriented library, an invocation is commonly preceded and succeeded by state-related operations (*e.g.*, updating the pointer to the marshaled data) and various checks to test the consistency of argument and return values. Given, an XDR description (*i.e.*, program), the RPCGEN compiler directly includes these operations when generating the marshaling code.

## 5.3 Domain-Specific Optimizations

Studying a program family is a valuable strategy to either identify domain-specific optimizations performed by expert programmers or to conceive new ones.

In the device driver case, optimization efforts mainly aim to minimize the communications with the device because of their cost. As a result, programmers try to update as many bit segments of a register as possible before issuing a write, instead of updating bit segments individually. The Devil compiler pursues the same goal and includes a dependency analysis to group device communications.

In the context of the marshaling process, one could conceive an optimization aimed to eliminate buffer overflow checks. This would entail the definition of a specific analysis to statically calculate values of buffer pointers in the case of fixed-length data. Based on this information, a code generator could then select the appropriate version of a marshaling instruction (*i.e.*, with or without a buffer overflow check). Such domain-specific optimization was automatically performed by program specialization and produced a significant speedup [15].

## 6 Assessing a DSL w.r.t. a Program Family

The evaluation of a DSL is a critical step toward giving it recognition in its target domain and thus contributing to its adoption. The program family plays a key role to assess a DSL. First, it enables to evaluate performance of DSL programs. Second, it can be used as a basis to assess the improved robustness resulting from the dedicated nature of the DSL. Third, it can be used to assess the conciseness of DSL programs. We do not discuss other, more subjective evaluation criteria, like ease of programming, abstraction level, readability, maintainability and usability.

### 6.1 Performance

In many areas, performance is a critical measure to assess the value of a DSL. At the very least, one should show that using a DSL does not entail any performance penalty. At most, it should demonstrate that its dedicated nature enables optimizations that are beyond the reach of compilers for GPLs.

Device drivers represent a typical domain where efficiency is critical for the overall system performance. As such, the recognition gained by Devil in the operating systems community, significantly relied on an extensive measurement analysis that showed that Devil-generated code did not induce significant execution overhead [8].

### 6.2 Robustness

Robustness can also be an important criterion to assess a DSL. For critical parts of a system, it is a significant contribution to demonstrate that a DSL enables bugs to be detected as early as possible during the development process. An interesting approach to assessing robustness is based on a mutation analysis [16]. In brief, this analysis consists of defining mutation rules to introduce errors in programs, while preserving its syntactic correctness. Then mutated programs are compiled and executed to measure how many errors are detected statically and/or dynamically.

In the device driver case, we compared the robustness of C mutated programs and Devil mutated programs. Experimental measurements showed that a Devil program is up to 6 times less prone to errors than C code.

As explained by Consel and Marlet [6], the improved robustness comes from both language restrictions and extra declarations. Both aspects contribute to enable static and/or dynamic checking of critical properties.

### 6.3 Conciseness

Comparing the conciseness of DSL programs to equivalent programs written in a GPL is not necessarily an easy task to do. The syntax of a DSL (*e.g.*, declarative) may drastically differ from that of a GPL (*e.g.*, imperative). Counting the number of lines or characters can thus be misleading. In our experience, a meaningful measure is the number of words. Yet, a comparison needs to be completed with a close examination of both sources to make sure that no unforeseen program aspects can distort measurements.

In practice, we observe that the more narrow the program family of a DSL the more concise the DSL programs. This observation is illustrated by the device driver domain. As mentioned earlier, our first study in this domain aimed to develop a DSL, named GAL, specifically targeting graphic display adaptors for the X11 server [13]. Then, a second study widened the scope of the target domain by addressing the communication layer of a device driver with Devil. On the one hand, the narrow scope of computations modeled by GAL leads to very concise programs that could be seen as a rich form of structured parameters. Experiments have shown that GAL programs are at least 9 times smaller than existing equivalent C device drivers [13]. On the other hand, the more general nature of Devil and the extra declarations provided by the programmer (*e.g.*, types of bit segments) do not allow similar conciseness. In fact, the size of a Devil program is similar to the size of its equivalent C program.

## 7 Conclusion

In practice, the design of a DSL is a very subjective process and has a lot of similarities with a craft. This situation is a key obstacle to spreading the DSL approach more widely. Methodologies to analyze a program family are quite general and do not specifically target the design of a DSL.

This paper advocates an approach to designing of a DSL tightly linked with the notion of program family. A program family is a concrete basis from which key properties can be extracted and fuel the design of a DSL. A program family naturally gives rise to a library, that, in turn, suggests domain-specific operations, domain-specific values and a domain-specific state. These elements contribute to define a domain-specific abstract machine. This abstract machine, combined with common program patterns from the program family, form the main ingredients to design a DSL. Lastly, a program family can be used to assess a DSL. In our experience, it produces objective measurements of the potential benefits of a DSL.

Unfortunately, in some cases, there are no program family to start with. This situation occurs when a new domain is defined. It happened, for example, when researchers studied the programmability of network routers to accelerate the development and deployment of new protocols [17]. Like other programming language researchers, we initiated the development of a DSL for this new domain. Our language, named Plan-P, aimed to safely program application-specific protocols [18]. Because routers were then mostly closed systems, there were very few programs to compare Plan-P programs against. As a result, the design of Plan-P relied on devising prospective applications. Even in this context, it can still be interesting to develop these prospective applications and build more intuition as to the issues involved in developing members of the future program family.

## Acknowledgment

This work has been partly supported by the *Conseil Régional d'Aquitaine* under Contract 20030204003A.

## References

1. Atkins, D., Ball, T., Baran, T., Benedikt, A., Cox, C., Ladd, D., Mataga, P., Puchol, C., Ramming, J., Rehor, K., Tuckey, C.: Integrated web and telephone service creation. *The Bell Labs Technical Journal* **2** (1997) 18–35
2. Brabrand, C., Møller, A., Schwartzbach, M.: The <bigwig> project. *ACM Transactions on Internet Technology* **2** (2002)
3. Czarnecki, K., Eisenecker, U.: *Generative Programming*. Addison-Wesley (2000)
4. Arnold, B., van Deursen, A., Res, M.: An algebraic specification of a language describing financial products. In: *IEEE Workshop on Formal Methods Application in Software Engineering*. (1995) 6–13
5. Consel, C., Réveillère, L.: A DSL paradigm for domains of services: A study of communication services (2004) In this volume.
6. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. In Palamidessi, C., Glaser, H., Meinke, K., eds.: *Proceedings of the 10<sup>th</sup> International Symposium on Programming Language Implementation and Logic Programming*. Number 1490 in *Lecture Notes in Computer Science*, Pisa, Italy (1998) 170–194
7. Weiss, D.: Family-oriented abstraction specification and translation: the FAST process. In: *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS)*, Gaithersburg, Maryland, IEEE Press, Piscataway, NJ (1996) 14–22
8. Mérillon, F., Réveillère, L., Consel, C., Marlet, R., Muller, G.: Devil: An IDL for hardware programming. In: *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California (2000) 17–30
9. Réveillère, L., Mérillon, F., Consel, C., Marlet, R., Muller, G.: A DSL approach to improve productivity and safety in device drivers development. In: *Proceedings of the 15<sup>th</sup> IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, IEEE Computer Society Press (2000) 101–109

10. Réveillère, L., Muller, G.: Improving driver robustness: an evaluation of the Devil approach. In: The International Conference on Dependable Systems and Networks, Göteborg, Sweden, IEEE Computer Society (2001) 131–140
11. Sun Microsystem: NFS: Network file system protocol specification. RFC 1094, Sun Microsystem (1989)
12. Parnas, D.: On the design and development of program families. *IEEE Transactions on Software Engineering* **2** (1976) 1–9
13. Thibault, S., Marlet, R., Consel, C.: Domain-specific languages: from design to implementation – application to video device drivers generation. *IEEE Transactions on Software Engineering* **25** (1999) 363–377
14. Schmidt, D.: *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, Inc. (1986)
15. Muller, G., Marlet, R., Volanschi, E., Consel, C., Pu, C., Goel, A.: Fast, optimized Sun RPC using automatic program specialization. In: *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, IEEE Computer Society Press (1998)
16. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. *Computer* **11** (1978) 34–41
17. Wetherall, D.: Active network vision and reality: lessons from a capsule-based system. In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC (1999)
18. Thibault, S., Consel, C., Muller, G.: Safe and efficient active network programming. In: *17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana (1998) 135–143



# A Gentle Introduction to Multi-stage Programming<sup>\*</sup>

Walid Taha

Department of Computer Science, Rice University, Houston, TX, USA  
`taha@rice.edu`

**Abstract.** Multi-stage programming (MSP) is a paradigm for developing generic software that does not pay a runtime penalty for this generality. This is achieved through concise, carefully-designed language extensions that support runtime code generation and program execution. Additionally, type systems for MSP languages are designed to statically ensure that dynamically generated programs are type-safe, and therefore require no type checking after they are generated.

This hands-on tutorial is aimed at the reader interested in learning the basics of MSP practice. The tutorial uses a freely available MSP extension of OCaml called MetaOCaml, and presents a detailed analysis of the issues that arise in staging an interpreter for a small programming language. The tutorial concludes with pointers to various resources that can be used to probe further into related topics.

## 1 Introduction

Although program generation has been shown to improve code reuse, product reliability and maintainability, performance and resource utilization, and developer productivity, there is little support for *writing* generators in mainstream languages such as C or Java. Yet a host of basic problems inherent in program generation can be addressed effectively by a programming language designed specifically to support writing generators.

### 1.1 Problems in Building Program Generators

One of the simplest approaches to writing program generators is to represent the program fragments we want to generate as either strings or data types (“abstract syntax trees”). Unfortunately, both representations have disadvantages. With the string encoding, we represent the code fragment `f (x,y)` simply as `"f (x,y)"`. Constructing and combining fragments represented by strings can be done concisely. But there is no *automatically verifiable* guarantee that programs constructed in this manner are syntactically correct. For example, `"f (,y)"` can have the static type `string`, but this string is clearly *not* a syntactically correct program.

---

<sup>\*</sup> Supported by NSF ITR-0113569 and NSF CCR-0205542.

With the data type encoding the situation is improved, but the best we can do is ensure that any generated program is syntactically correct. We cannot use data types to ensure that generated programs are well-typed. The reason is that data types can represent context-free sets accurately, but usually not context sensitive sets. Type systems generally define context sensitive sets (of programs). Constructing data type values that represent trees can be a bit more verbose, but a quasi-quotation mechanism [1] can alleviate this problem and make the notation as concise as that of strings.

In contrast to the strings encoding, MSP languages statically ensure that any generator only produces syntactically well-formed programs. Additionally, statically typed MSP languages statically ensure that any generated program is also well-typed.

Finally, with both string and data type representations, ensuring that there are no name clashes or inadvertent variable captures *in the generated program* is the responsibility of the programmer. This is essentially the same problem that one encounters with the C macro system. MSP languages ensure that such inadvertent capture is not possible. We will return to this issue when we have seen one example of MSP.

## 1.2 The Three Basic MSP Constructs

We can illustrate how MSP addresses the above problems using MetaOCaml [2], an MSP extension of OCaml [9]. In addition to providing traditional imperative, object-oriented, and functional constructs, MetaOCaml provides three constructs for staging. The constructs are called Brackets, Escape, and Run. Using these constructs, the programmer can change the order of evaluation of terms. This capability can be used to reduce the overall cost of a computation.

**Brackets** (written `.<...>.`) can be inserted around any expression to delay its execution. MetaOCaml implements delayed expressions by dynamically generating source code at runtime. While using the source code representation is not the only way of implementing MSP languages, it is the simplest. The following short interactive MetaOCaml session illustrates the behavior of Brackets<sup>1</sup>:

```
# let a = 1+2;;
val a : int = 3
# let a = .<1+2>.;;
val a : int code = .<1+2>.
```

Lines that start with `#` are what is entered by the user, and the following line(s) are what is printed back by the system. Without the Brackets around `1+2`, the addition is performed right away. With the Brackets, the result is a piece of code representing the program `1+2`. This code fragment can either be used as part of another, bigger program, or it can be compiled and executed.

<sup>1</sup> Some versions of MetaOCaml developed after December 2003 support environment classifiers [21]. For these systems, the type `int code` is printed as `('a,int) code`. To follow the examples in this tutorial, the extra parameter `'a` can be ignored.

In addition to delaying the computation, Brackets are also reflected in the type. The type in the last declaration is `int code`. The type of a code fragment reflects the type of the value that such code should produce when it is executed. Statically determining the type of the generated code allows us to avoid writing generators that produce code that cannot be typed. The code type constructor distinguishes delayed values from other values and prevents the user from accidentally attempting unsafe operations (such as `1 + .<5>.`).

**Escape** (written `.~...`) allows the combination of smaller delayed values to construct larger ones. This combination is achieved by “splicing-in” the argument of the Escape in the context of the surrounding Brackets:

```
# let b = .<.~a * .~a >. ;;
val b : int code = .<(1 + 2) * (1 + 2)>.
```

This declaration binds `b` to a new delayed computation  $(1+2)*(1+2)$ .

**Run** (written `.!...`) allows us to compile and execute the dynamically generated code without going outside the language:

```
# let c = .! b;;
val c : int = 9
```

Having these three constructs as part of the programming language makes it possible to use runtime code generation and compilation as part of any library subroutine. In addition to not having to worry about generating temporary files, static type systems for MSP languages can assure us that no runtime errors will occur in these subroutines (c.f. [17]). Not only can these type systems exclude generation-time errors, but they can also ensure that generated programs are both syntactically well-formed and well-typed. Thus, the ability to statically type-check the safety of a computation is not lost by staging.

### 1.3 Basic Notions of Equivalence

As an aside, there are two basic equivalences that hold for the three MSP constructs [18]:

$$\begin{aligned} .\sim .<e>. &= e \\ .! .<e>. &= e \end{aligned}$$

Here, a value  $v$  can include usual values such as integers, booleans, and lambdas, as well as Bracketed terms of the form `.<e>.`. In the presentation above, we use  $e$  for an expression where all Escapes are enclosed by a matching set of Brackets. The rules for Escape and Run are identical. The distinction between the two constructs is in the notion of values: the expression in the value `.<e>.` cannot contain Escapes that are not locally surrounded by their own Brackets. An expression  $e$  is unconstrained as to where Run can occur.

**Avoiding accidental name capture.** Consider the following staged function:

```
# let rec h n z = if n=0 then z
                  else .<(fun x -> .~(h (n-1) .<x+ .~z>..)) n>.;;
val h : int -> int code -> int code = <fun>
```

If we erase the annotations (to get the “unstaged” version of this function) and apply it to 3 and 1, we get 7 as the answer. If we apply the staged function above to 3 and `.<1>.`, we get the following term:

```
.<(fun x_1 -> (fun x_2 -> (fun x_3 -> x_3 + (x_2 + (x_1 + 1)))) 1) 2) 3>.
```

Whereas the source code only had `fun x -> ...` inside Brackets, this code fragment was generated three times, and each time it produced a different `fun x_i -> ...` where `i` is a different number each time. If we run the generated code above, we get 7 as the answer. We view it as a highly desirable property that the results generated by staged programs are related to the results generated by the unstaged program. The reader can verify for herself that if the `xs` were not renamed and we allowed variable capture, the answer of running the staged program would be different from 7. Thus, automatic renaming of bound variables is not so much a feature; rather, it is the absence of renaming that seems like a bug.

## 1.4 Organization of This Paper

The goal of this tutorial is to familiarize the reader with the basics of MSP. To this end, we present a detailed example of how MSP can be used to build staged interpreters. The practical appeal of staged interpreters lies in that they can be almost as simple as interpreter-based language implementations and at the same time be as efficient as compiler-based ones. To this end, Section 2 briefly describes an idealized method for developing staged programs in a programming language that provides MSP constructs. The method captures a process that a programmer iteratively applies while developing a staged program. This method will be used repeatedly in examples in the rest of the paper. Section 3 constitutes the technical payload of the paper, and presents a series of more sophisticated interpreters and staged interpreters for a toy programming language. This section serves both to introduce key issues that arise in the development of a staged interpreter (such as error handling and binding-time improvements), and to present a realistic example of a series of iterative refinements that arise in developing a satisfactory staged interpreter. Section 4 concludes with a brief overview of additional resources for learning more about MSP.

## 2 How Do We Write MSP Programs?

Good abstraction mechanisms can help the programmer write more concise and maintainable programs. But if these abstraction mechanisms degrade performance, they will not be used. The primary goal of MSP is to help the programmer reduce the runtime overhead of sophisticated abstraction mechanisms. Our

hope is that having such support will allow programmers to write higher-level and more reusable programs.

## 2.1 A Basic Method for Building MSP Programs

An idealized method for building MSP programs can be described as follows:

1. A single-stage program is developed, implemented, and tested.
2. The organization and data-structures of the program are studied to ensure that they can be used in a staged manner. This analysis may indicate a need for “factoring” some parts of the program and its data structures. This step can be critical step toward effective MSP. Fortunately, it has been thoroughly investigated in the context of partial evaluation where it is known as *binding-time engineering* [7].
3. Staging annotations are introduced to explicitly specify the evaluation order of the various computations of the program. The staged program may then be tested to ensure that it achieves the desired performance.

The method described above, called *multi-stage programming with explicit annotations* [22], can be summarized by the slogan:

A Staged Program = A Conventional Program + Staging Annotations

We consider the method above to be idealized because, in practice, it is useful to iterate through steps 1-3 in order to determine the program that is most suitable for staging. This iterative process will be illustrated in the rest of this paper. But first, we consider a simpler example where the method can be applied directly.

## 2.2 A Classic Example

A common source of performance overhead in generic programs is the presence of parameters that do not change very often, but nevertheless cause our programs to repeatedly perform the work associated with these inputs. To illustrate, we consider the following classic function in MetaOCaml: [7]

```
let rec power (n, x) =
  match n with
  0 -> 1 | n -> x * (power (n-1, x));;
```

This function is generic in that it can be used to compute  $x$  raised to *any* nonnegative exponent  $n$ . While it is convenient to use generic functions like `power`, we often find that we have to pay a price for their generality. Developing a good understanding of the source of this performance penalty is important, because it is exactly what MSP will help us eliminate. For example, if we need to compute the second power often, it is convenient to define a special function:

```
let power2 (x) = power (2,x);;
```

In a functional language, we can also write it as follows:

```
let power2 = fun x -> power (2,x);;
```

Here, we have taken away the formal parameter `x` from the left-hand side of the equality and replaced it by the equivalent “`fun x ->`” on the right-hand side. To use the function `power2`, all we have to do is to apply it as follows:

```
let answer = power2 (3);;
```

The result of this computation is 9. But notice that every time we apply `power2` to some value `x` it calls the `power` function with parameters `(2,x)`. And even though the first argument will always be 2, evaluating `power (2,x)` will always involve calling the function recursively two times. This is an undesirable overhead, because we *know* that the result can be more efficiently computed by multiplying `x` by itself. Using only unfolding and the definition of `power`, we know that the answer can be computed more efficiently by:

```
let power2 = fun x -> 1*x*x;;
```

We also do not want to write this by hand, as there may be many other specialized power functions that we wish to use. So, can we *automatically* build such a program?

In an MSP language such as MetaOCaml, all we need to do is to *stage* the power function by annotating it:

```
let rec power (n, x) =
  match n with
  0 -> .<1>. | n -> .<~x * ~(power (n-1, x))>.;;
```

This function still takes two arguments. The second argument is no longer an integer, but rather, a *code of type integer*. The return type is also changed. Instead of returning an integer, this function will return a code of type integer. To match this return type, we insert Brackets around 1 in the first branch on the third line. By inserting Brackets around the multiplication expression, we now return a code of integer instead of just an integer. The Escape around the recursive call to `power` means that it is performed immediately.

The staging constructs can be viewed as “annotations” on the original program, and are fairly unobtrusive. Also, we are able to type check the code both outside and inside the Brackets in essentially the same way that we did before. If we were using strings instead of Brackets, we would have to sacrifice static type checking of delayed computations.

After annotating `power`, we have to annotate the uses of `power`. The declaration of `power2` is annotated as follows:

```
let power2 = .! .<fun x -> ~(power (2,<x>))>.;;
```

Evaluating the application of the Run construct will compile and if execute its argument. Notice that this declaration is essentially the same as what we used to define `power2` before, except for the staging annotations. The annotations say that we wish to construct the code for a function that takes one argument (`fun x ->`). We also do not spell out what the function itself should do; rather, we use the Escape construct (`.~`) to make a call to the staged `power`. The result

of evaluating this declaration behaves exactly as if we had defined it “by hand” to be `fun x -> 1*x*x`. Thus, it will behave exactly like the first declaration of `power2`, but it will run as though we had written the specialized function by hand. The staging constructs allowed us to eliminate the runtime overhead of using the generic power function.

### 3 Implementing DSLs Using Staged Interpreters

An important application for MSP is the implementation of domain-specific languages (DSLs) [4]. Languages can be implemented in a variety of ways. For example, a language can be implemented by an interpreter or by a compiler. Compiled implementations are often orders of magnitude faster than interpreted ones, but compilers have traditionally required significant expertise and took orders of magnitude more time to implement than interpreters. Using the method outlined in Section 2, we can start by first writing an interpreter for a language, and then stage it to get *a staged interpreter*. Such staged interpreters can be as simple as the interpreter we started with and at the same time have performance comparable to that of a compiler. This is possible because the staged interpreter becomes effectively a *translator* from the DSL to the host language (in this case OCaml). Then, by using MetaOCaml’s `Run` construct, the total effect is in fact a function that takes DSL programs and produces machine code<sup>2</sup>.

To illustrate this approach, this section investigates the implementation of a toy programming language that we call `lint`. This language supports integer arithmetic, conditionals, and recursive functions. A series of increasingly more sophisticated interpreters and staged interpreters are used to give the reader a hands-on introduction to MSP. The complete code for the implementations described in this section is available online [10].

#### 3.1 Syntax

The syntax of expressions in this language can be represented in OCaml using the following data type:

```
type exp = Int of int | Var of string | App of string * exp
        | Add of exp * exp | Sub of exp * exp
        | Mul of exp * exp | Div of exp * exp | Ifz of exp * exp * exp

type def = Declaration of string * string * exp
type prog = Program of def list * exp
```

Using the above data types, a small program that defines the factorial function and then applies it to 10 can be concisely represented as follows:

---

<sup>2</sup> If we are using the MetaOCaml native code compiler. If we are using the bytecode compiler, the composition produces bytecode.

```

Program ([Declaration
  ("fact","x", Ifz(Var "x",
    Int 1,
    Mul(Var"x",
      (App ("fact", Sub(Var "x",Int 1))))))
],
  App ("fact", Int 10))

```

OCaml lex and yacc can be used to build parsers that take textual representations of such programs and produce abstract syntax trees such as the above. In the rest of this section, we focus on what happens after such an abstract syntax tree has been generated.

### 3.2 Environments

To associate variable and function names with their values, an interpreter for this language will need a notion of an environment. Such an environment can be conveniently implemented as a function from names to values. If we look up a variable and it is not in the environment, we will raise an exception (let's call it **Yikes**). If we want to extend the environment (which is just a function) with an association from the name **x** to a value **v**, we simply return a new environment (a function) which first tests to see if its argument is the same as **x**. If so, it returns **v**. Otherwise, it looks up its argument in the original environment. All we need to implement such environments is the following:

```

exception Yikes

(* env0, fenv : 'a -> 'b *)

let env0 = fun x -> raise Yikes          let fenv0 = env0

(* ext : ('a -> 'b) -> 'a -> 'b -> 'a -> 'b *)

let ext env x v = fun y -> if x=y then v else env y

```

The types of all three functions are polymorphic. Type variables such as **'a** and **'b** are implicitly universally quantified. This means that they can be later instantiated to more specific types. Polymorphism allows us, for example, to define the initial function environment **fenv0** as being exactly the same as the initial variable environment **env0**. It will also allow us to use the same function **ext** to extend both kinds of environments, even when their types are instantiated to the more specific types such as:

```

env0 : string -> int                fenv0 : string -> (int -> int)

```

### 3.3 A Simple Interpreter

Given an environment binding variable names to their runtime values and function names to values (these will be the formal parameters **env** and **fenv**, respectively), an interpreter for an expression **e** can be defined as follows:



```

(* eval1 : exp -> (string -> int) -> (string -> int -> int) -> int *)

let rec eval1 e env fenv =
match e with
| Int i -> i
| Var s -> env s
| App (s,e2) -> (fenv s)(eval1 e2 env fenv)
| Add (e1,e2) -> (eval1 e1 env fenv)+(eval1 e2 env fenv)
| Sub (e1,e2) -> (eval1 e1 env fenv)-(eval1 e2 env fenv)
| Mul (e1,e2) -> (eval1 e1 env fenv)*(eval1 e2 env fenv)
| Div (e1,e2) -> (eval1 e1 env fenv)/(eval1 e2 env fenv)
| Ifz (e1,e2,e3) -> if (eval1 e1 env fenv)=0
                        then (eval1 e2 env fenv)
                        else (eval1 e3 env fenv)

```

This interpreter can now be used to define the interpreter for declarations and programs as follows:

```

(* peval1 : prog -> (string -> int) -> (string -> int -> int) -> int *)

let rec peval1 p env fenv=
  match p with
  | Program ([],e) -> eval1 e env fenv
  | Program (Declaration (s1,s2,e1)::tl,e) ->
    let rec f x = eval1 e1 (ext env s2 x) (ext fenv s1 f)
    in peval1 (Program(tl,e)) env (ext fenv s1 f)

```

In this function, when the list of declarations is empty (`[]`), we simply use `eval1` to evaluate the body of the program. Otherwise, we recursively interpret the list of declarations. Note that we also use `eval1` to interpret the body of function declarations. It is also instructive to note the three places where we use the environment extension function `ext` on both variable and function environments.

The above interpreter is a complete and concise specification of what programs in this language should produce when they are executed. Additionally, this style of writing interpreters follows quite closely what is called the denotational style of specifying semantics, which can be used to specify a wide range of programming languages. It is reasonable to expect that a software engineer can develop such implementations in a short amount of time.

**Problem: The cost of abstraction.** If we evaluate the factorial example given above, we will find that it runs about 20 times slower than if we had written this example directly in OCaml. The main reason for this is that the interpreter repeatedly traverses the abstract syntax tree during evaluation. Additionally, environment lookups in our implementation are not constant-time.

### 3.4 The Simple Interpreter Staged

MSP allows us to keep the conciseness and clarity of the implementation given above and also eliminate the performance overhead that traditionally we would

have had to pay for using such an implementation. The overhead is avoided by staging the above function as follows:

```
(* eval2 : exp -> (string -> int code) -> (string -> (int -> int) code)
   -> int code *)

let rec eval2 e env fenv =
  match e with
  | Int i -> .<i>.
  | Var s -> env s
  | App (s,e2) -> .<~(fenv s).~(eval2 e2 env fenv)>.
  ...
  | Div (e1,e2)-> .<~(eval2 e1 env fenv)/ ~.(eval2 e2 env fenv)>.
  | Ifz (e1,e2,e3) -> .<if ~(eval2 e1 env fenv)=0
                        then ~(eval2 e2 env fenv)
                        else ~(eval2 e3 env fenv)>.

(* peval2 : prog -> (string -> int code) -> (string -> (int -> int) code)
   -> int code *)

let rec peval2 p env fenv=
  match p with
  | Program ([],e) -> eval2 e env fenv
  | Program (Declaration (s1,s2,e1)::t1,e) ->
    .<let rec f x = ~.(eval2 e1 (ext env s2 .<x>.)
                        (ext fenv s1 .<f>.))
      in ~.(peval2 (Program(t1,e)) env (ext fenv s1 .<f>.))>.
```

If we apply `peval2` to the abstract syntax tree of the factorial example (given above) and the empty environments `env0` and `fenv0`, we get back the following code fragment:

```
.<let rec f = fun x -> if x = 0 then 1 else x * (f (x - 1)) in (f 10)>.
```

This is exactly the same code that we would have written by hand for that specific program. Running this program has exactly the same performance as if we had written the program directly in OCaml.

The staged interpreter is a function that takes abstract syntax trees and produces MetaOCaml programs. This gives rise to a simple but often overlooked fact [19,20]:

A staged interpreter is a translator.

It is important to keep in mind that the above example is quite simplistic, and that further research is needed to show that we can apply MSP to realistic programming languages. Characterizing what MSP cannot do is difficult, because of the need for technical expressivity arguments. We take the more practical approach of explaining what MSP can do, and hope that this gives the reader a working understanding of the scope of this technology.

### 3.5 Error Handling

Returning to our example language, we will now show how direct staging of an interpreter does not always yield satisfactory results. Then, we will give an example of the wide range of techniques that can be used in such situations.

A generally desirable feature of programming languages is error handling. The original implementation of the interpreter uses the division operation, which can raise a divide-by-zero exception. If the interpreter is part of a bigger system, we probably want to have finer control over error handling. In this case, we would modify our original interpreter to perform a check before a division, and return a special value `None` if the division could not be performed. Regular values that used to be simply `v` will now be represented by `Some v`. To do this, we will use the following standard datatype:

```
type 'a option = None | Just of 'a;;
```

Now, such values will have to be propagated and dealt with everywhere in the interpreter:

```
(* eval3 : exp -> (string -> int) -> (string -> int -> int option)
   -> int option *)

let rec eval3 e env fenv =
match e with
  Int i -> Some i
| Var s -> Some (env s)
| App (s,e2) -> (match (eval3 e2 env fenv) with
                  Some x -> (fenv s) x
                  | None   -> None)
| Add (e1,e2) -> (match (eval3 e1 env fenv, eval3 e2 env fenv)
                  with (Some x, Some y) -> Some (x+y)
                      | _ -> None) ...
| Div (e1,e2) -> (match (eval3 e1 env fenv, eval3 e2 env fenv)
                  with (Some x, Some y) ->
                      if y=0 then None
                      else Some (x/y)
                      | _ -> None)
| Ifz (e1,e2,e3) -> (match (eval3 e1 env fenv) with
                     Some x -> if x=0 then (eval3 e2 env fenv)
                               else (eval3 e3 env fenv)
                     | None   -> None)
```

Compared to the unstaged interpreter, the performance overhead of adding such checks is marginal. But what we really care about is the staged setting. Staging `eval3` yields:

```
(* eval4 : exp -> (string -> int code)
   -> (string -> (int -> int option) code)
   -> (int option) code *)

let rec eval4 e env fenv =
match e with
```

```

Int i -> .<Some i>.
| Var s -> .<Some .~(env s)>.
| App (s,e2) -> .<(match .~(eval4 e2 env fenv) with
    Some x -> .~(fenv s) x
    | None   -> None)>.
| Add (e1,e2) -> .<(match (.~(eval4 e1 env fenv),
    .~(eval4 e2 env fenv)) with
    (Some x, Some y) -> Some (x+y)
    | _ -> None)>. ...
| Div (e1,e2) -> .<(match (.~(eval4 e1 env fenv),
    .~(eval4 e2 env fenv)) with
    (Some x, Some y) ->
        if y=0 then None
        else Some (x/y)
    | _ -> None)>.
| Ifz (e1,e2,e3) -> .<(match .~(eval4 e1 env fenv) with
    Some x -> if x=0 then
        .~(eval4 e2 env fenv)
        else
        .~(eval4 e3 env fenv)
    | None   -> None)>.

```

**Problem: The cost of error handling.** Unfortunately, the performance of code generated by this staged interpreter is typically 4 times slower than the first staged interpreter that had no error handling. The source of the runtime cost becomes apparent when we look at the generated code:

```

.<let rec f =
  fun x ->
    (match (Some (x)) with
    Some (x) ->
      if (x = 0) then (Some (1))
      else
        (match
          ((Some (x)),
          (match
            (match ((Some (x)), (Some (1))) with
              (Some (x), Some (y)) ->
                (Some ((x - y)))
              | _ -> (None)) with
            Some (x) -> (f x)
            | None -> (None))) with
          (Some (x), Some (y)) ->
            (Some ((x * y)))
          | _ -> (None))
        | None -> (None)) in
    (match (Some (10)) with
    Some (x) -> (f x)
    | None -> (None))>.

```

The generated code is doing much more work than before, because at every operation we are checking to see if the values we are operating with are proper values or not. Which branch we take in every `match` is determined by the explicit form of the value being matched.

**Half-solutions.** One solution to such a problem is certainly adding a pre-processing analysis. But if we can avoid generating such inefficient code in the first place it would save the time wasted both in generating these unnecessary checks and in performing the analysis. More importantly, with an analysis, we may never be certain that all unnecessary computation is eliminated from the generated code.

### 3.6 Binding-Time Improvements

The source of the problem is the `if` statement that appears in the interpretation of `Div`. In particular, because `y` is bound inside Brackets, we cannot perform the test `y=0` while we are building for these Brackets. As a result, we cannot immediately determine if the function should return a `None` or a `Some` value. This affects the type of the whole staged interpreter, and effects the way we interpret all programs even if they do not contain a use of the `Div` construct.

The problem can be avoided by what is called a *binding-time improvement* in the partial evaluation literature [7]. It is essentially a transformation of the program that we are staging. The goal of this transformation is to allow better staging. In the case of the above example, one effective binding time improvement is to rewrite the interpreter in continuation-passing style (CPS) [5], which produces the following code:

```
(* eval5 : exp -> (string -> int) -> (string -> int -> int)
   -> (int option -> 'a) -> 'a *)

let rec eval5 e env fenv k =
match e with
  Int i -> k (Some i)
| Var s -> k (Some (env s))
| App (s,e2) -> eval5 e2 env fenv
                (fun r -> match r with
                  Some x -> k (Some ((fenv s) x))
                  | None   -> k None)
| Add (e1,e2) -> eval5 e1 env fenv
                (fun r ->
                  eval5 e2 env fenv
                    (fun s -> match (r,s) with
                      (Some x, Some y) -> k (Some (x+y))
                      | _ -> k None)) ...)
| Div (e1,e2) -> eval5 e1 env fenv
                (fun r ->
                  eval5 e2 env fenv
```

```

      (fun s -> match (r,s) with
        (Some x, Some y) ->
          if y=0 then k None
            else k (Some (x/y))
        | _ -> k None)) ...

(* pevalK5 : prog -> (string -> int) -> (string -> int -> int)
   -> (int option -> int) -> int *)

let rec pevalK5 p env fenv k =
  match p with
  | Program ([],e) -> eval5 e env fenv k
  | Program (Declaration (s1,s2,e1)::tl,e) ->
    let rec f x = eval5 e1 (ext env s2 x) (ext fenv s1 f) k
    in pevalK5 (Program(tl,e)) env (ext fenv s1 f) k

exception Div_by_zero;;

(* peval5 : prog -> (string -> int) -> (string -> int -> int) -> int *)

let peval5 p env fenv =
  pevalK5 p env fenv (function Some x -> x
                        | None -> raise Div_by_zero)

```

In the unstaged setting, we can use the CPS implementation to get the same functionality as the direct-style implementation. But note that the two algorithms are *not the same*. For example, performance of the CPS interpreter is in fact worse than the previous one. But when we try to stage the new interpreter, we find that we can do something that we could not do in the direct-style interpreter. In particular, the CPS interpreter can be staged as follows:

```

(* eval6 : exp -> (string -> int code) -> (string -> (int -> int) code)
   -> (int code option -> 'b code) -> 'b code *)

let rec eval6 e env fenv k =
  match e with
  | Int i -> k (Some .<i>.)
  | Var s -> k (Some (env s))
  | App (s,e2) -> eval6 e2 env fenv
    (fun r -> match r with
      Some x -> k (Some .<.<~(fenv s) .~x>.)
      | None -> k None)
  | Add (e1,e2) -> eval6 e1 env fenv
    (fun r ->
      eval6 e2 env fenv
      (fun s -> match (r,s) with
        (Some x, Some y) ->
          k (Some .<.<~x + .~y>.)
        | _ -> k None)) ...
  | Div (e1,e2) -> eval6 e1 env fenv
    (fun r ->

```

```

eval6 e2 env fenv
  (fun s -> match (r,s) with
    (Some x, Some y) ->
      .<if .~y=0 then .~(k None)
        else .~(k (Some .<~x / ~y>))>.
    | _ -> k None))
| Ifz (e1,e2,e3) -> eval6 e1 env fenv
  (fun r -> match r with
    Some x -> .<if .~x=0 then
      .~(eval6 e2 env fenv k)
    else
      .~(eval6 e3 env fenv k)>.
    | None -> k None)

(* peval6 : prog -> (string -> int code) -> (string -> (int -> int) code)
   -> int code *)

let peval6 p env fenv =
  pevalK6 p env fenv (function Some x -> x
    | None -> .<raise Div_by_zero>.)

```

The improvement can be seen at the level of the type of `eval6`: the `option` type occurs outside the `code` type, which suggests that it can be eliminated in the first stage. What we could not do before is to Escape the application of the continuation to the branches of the `if` statement in the `Div` case. The extra advantage that we have when staging a CPS program is that we are applying the continuation multiple times, which is essential for performing the computation in the branches of an `if` statement. In the unstaged CPS interpreter, the continuation is always applied exactly once. Note that this is the case even in the `if` statement used to interpret `Div`: The continuation does *occur* twice (once in each branch), but only one branch is ever taken when we evaluate this statement. But in the the staged interpreter, the continuation is indeed duplicated and applied multiple times<sup>3</sup>.

The staged CPS interpreter generates code that is exactly the same as what we got from the first interpreter as long as the program we are interpreting does not use the division operation. When we use division operations (say, if we replace the code in the body of the fact example with `fact (20/2)`) we get the following code:

```

.<let rec f =
  fun x -> if (x = 0) then 1 else (x * (f (x - 1)))
in if (2 = 0) then (raise (Div_by_zero)) else (f (20 / 2))>.

```

---

<sup>3</sup> This means that converting a program into CPS can have disadvantages (such as a computationally expensive first stage, and possibly code duplication). Thus, CPS conversion must be used judiciously [8].

### 3.7 Controlled Inlining

So far we have focused on eliminating unnecessary work from generated programs. Can we do better? One way in which MSP can help us do better is by allowing us to unfold function declarations for a fixed number of times. This is easy to incorporate into the first staged interpreter as follows:

```
(* eval7 : exp -> (string -> int code)
   -> (string -> int code -> int code) -> int code *)

let rec eval7 e env fenv =
match e with
... most cases the same as eval2, except
| App (s,e2) -> fenv s (eval7 e2 env fenv)

(* repeat : int -> ('a -> 'a) -> 'a -> 'a *)

let rec repeat n f =
  if n=0 then f else fun x -> f (repeat (n-1) f x)

(* peval7 : prog -> (string -> int code)
   -> (string -> int code -> int code) -> int code *)

let rec peval7 p env fenv=
  match p with
  Program ([],e) -> eval7 e env fenv
  | Program (Declaration (s1,s2,e1)::tl,e) ->
    .<let rec f x =
      .~(let body cf x =
          eval7 e1 (ext env s2 x) (ext fenv s1 cf) in
          repeat 1 body (fun y -> .<f .~y>.) .<x>.)
      in .~(peval7 (Program(tl,e)) env
          (ext fenv s1 (fun y -> .<f .~y>))))>.
```

The code generated for the factorial example is as follows:

```
.<let rec f =
  fun x ->
    if (x = 0) then 1
    else
      (x*(if ((x-1)=0) then 1 else (x-1)*(f ((x-1)-1))))
  in (f 10)>.
```

This code can be expected to be faster than that produced by the first staged interpreter, because only one function call is needed for every two iterations.

**Avoiding Code Duplication.** The last interpreter also points out an important issue that the multi-stage programmer must pay attention to: code duplication. Notice that the term `x-1` occurs three times in the generated code. In the result of the first staged interpreter, the subtraction only occurred once.



The duplication of this term is a result of the inlining that we perform on the body of the function. If the argument to a recursive call was even bigger, then code duplication would have a more dramatic effect both on the time needed to compile the program and the time needed to run it.

A simple solution to this problem comes from the partial evaluation community: we can generate `let` statements that replace the expression about to be duplicated by a simple variable. This is only a small change to the staged interpreter presented above:

```
let rec eval8 e env fenv =
match e with
... same as eval7 except for
| App (s,e2) -> .<let x= .~(eval8 e2 env fenv)
               in .~(fenv s .<x>.)>. ...
```

Unfortunately, in the current implementation of MetaOCaml this change does not lead to a performance improvement. The most likely reason is that the bytecode compiler does not seem to perform `let`-floating. In the native code compiler for MetaOCaml (currently under development) we expect this change to be an improvement.

Finally, both error handling and inlining can be combined into the same implementation:

```
(* eval9: exp -> (string -> int code) -> (string -> int code -> int code)
   -> (int code option -> 'b code) -> 'b code *)

let rec eval9 e env fenv k =
match e with
... same as eval6, except
| App (s,e2) -> eval9 e2 env fenv
               (fun r -> match r with
                Some x -> k (Some ((fenv s) x))
                | None  -> k None)

(* pevalK9 : prog -> (string -> int code)
   -> (string -> int code -> int code)
   -> (int code option -> int code) -> int code *)

let rec pevalK9 p env fenv k =
match p with
Program ([],e) -> eval9 e env fenv k
|Program (Declaration (s1,s2,e1)::tl,e) ->
  .<let rec f x =
    .~(let body cf x =
      eval9 e1 (ext env s2 x) (ext fenv s1 cf) k in
      repeat 1 body (fun y -> .<f .~y>.) .<x>.)
    in .~(pevalK9 (Program(tl,e)) env
      (ext fenv s1 (fun y -> .<f .~y>.) k)>.
```

### 3.8 Measuring Performance in MetaOCaml

MetaOCaml provides support for collecting performance data, so that we can empirically verify that staging does yield the expected performance improvements. This is done using three simple functions. The first function must be called before we start collecting timings. It is called as follows:

```
Trx.init_times ();;
```

The second function is invoked when we want to gather timings. Here we call it twice on both `power2 (3)` and `power2' (3)` where `power2'` is the staged version:

```
Trx.timenew "Normal" (fun () ->(power2 (3)));;
Trx.timenew "Staged" (fun () ->(power2' (3)));;
```

Each call to `Trx.timenew` causes the argument passed last to be run as many times as this system needs to gather a reliable timing. The quoted strings simply provide hints that will be printed when we decide to print the summary of the timings. The third function prints a summary of timings:

```
Trx.print_times ();;
```

The following table summarizes timings for the various functions considered in the previous section<sup>4</sup>:

Program	Description of Interpreter	Fact10	Fib20
<i>(none)</i>	OCaml implementations	100%	100%
eval1	Simple	1,570%	1,736%
eval2	Simple staged	100%	100%
eval3	Error handling (EH)	1,903%	2,138%
eval4	EH staged	417%	482%
eval5	CPS, EH	2,470%	2,814%
eval6	CPS, EH, staged	100%	100%
eval7	Inlining, staged	87%	85%
eval8	Inlining, no duplication, staged	97%	97%
eval9	Inlining, CPS, EH, staged	90%	85%

Timings are normalized relative to handwritten OCaml versions of the DSL programs that we are interpreting (`Fact10` and `Fib20`). Lower percentages mean faster execution. This kind of normalization does not seem to be commonly used in the literature, but we believe that it has an important benefit. In particular, it serves to emphasize the fact that the performance of the resulting specialized programs should really be compared to specialized hand-written programs. The fact that `eval9` is more than 20 times faster than `eval3` is often irrelevant to a programmer who rightly considers `eval3` to be an impractically inefficient implementation. As we mentioned earlier in this paper, the goal of MSP is to remove unnecessary runtime costs associated with abstractions, and identifying the right reference point is essential for knowing when we have succeeded.

<sup>4</sup> System specifications: MetaOCaml bytecode interpreter for OCaml 3.07+2 running under Cygwin on a Pentium III machine, Mobile CPU, 1133MHz clock, 175 MHz bus, 640 MB RAM. Numbers vary when Linux is used, and when the native code compiler is used.

## 4 To Probe Further

The sequence of interpreters presented here is intended to illustrate the iterative nature of the process of exploring how to stage an interpreter for a DSL so that it can be turned into a satisfactory translator, which when composed with the Run construct of MetaOCaml, gives a compiler for the DSL.

A simplifying feature of the language studied here is that it is a first-order language with only one type, `int`. For richer languages we must define a datatype for values, and interpreters will return values of this datatype. Using such a datatype has an associated runtime overhead. Eliminating the overhead for such datatypes can be achieved using either a post-processing transformation called tag elimination [20] or MSP languages with richer static type systems [14]. MetaOCaml supports an experimental implementation of tag elimination [2].

The extent to which MSP is effective is often dictated by the surrounding environment in which an algorithm, program, or system is to be used. There are three important examples of situations where staging can be beneficial:

- We want to minimize total cost of all stages *for most inputs*. This model applies, for example, to implementations of programming languages. The cost of a simple compilation followed by execution is *usually* lower than the cost of interpretation. For example, the program being executed usually contains loops, which typically incur large overhead in an interpreted implementation.
- We want to minimize *a weighted average* of the cost of all stages. The weights reflect the relative frequency at which the result of a stage can be reused. This situation is relevant in many applications of symbolic computation. Often, solving a problem symbolically, and then graphing the solution at a thousand points can be cheaper than numerically solving the problem a thousand times. This cost model can make a symbolic approach worthwhile even when it is 100 times more expensive than a direct numerical one. (By symbolic computation we simply mean computation where free variables are values that will only become available at a later stage.)
- We want to minimize the cost of the *last stage*. Consider an embedded system where the *sin* function may be implemented as a large look-up table. The cost of constructing the table is not relevant. Only the cost of computing the function at run-time is. The same applies to optimizing compilers, which may spend an unusual amount of time to generate a high-performance computational library. The cost of optimization is often not relevant to the users of such libraries.

The last model seems to be the most commonly referenced one in the literature, and is often described as “there is ample time between the arrival of different inputs”, “there is a significant difference between the frequency at which the various inputs to a program change”, and “the performance of the program matters only after the arrival of its last input”.

Detailed examples of MSP can be found in the literature, including term-rewriting systems [17] and graph algorithms [12]. The most relevant example to

domain-specific program generation is on implementing a small imperative language [16]. The present tutorial should serve as good preparation for approaching the latter example.

An implicit goal of this tutorial is to prepare the reader to approach introductions to partial evaluation [3] and partial evaluation of interpreters in particular [6]. While these two works can be read without reading this tutorial, developing a full appreciation of the ideas they discuss requires either an understanding of the internals of partial evaluators or a basic grasp of multi-stage computation. This tutorial tries to provide the latter.

Multi-stage languages are both a special case of meta-programming languages and a generalization of multi-level and two-level languages. Taha [17] gives definitions and a basic classification for these notions. Sheard [15] gives a recent account of accomplishments and challenges in meta-programming. While multi-stage languages are “only” a special case of meta-programming languages, their specialized nature has its advantages. For example, it is possible to formally prove that they admit strong algebraic properties even in the untyped setting, but this is not the case for more general notions of meta-programming [18]. Additionally, significant advances have been made over the last six years in static typing for these languages (c.f. [21]). It will be interesting to see if such results can be attained for more general notions of meta-programming.

Finally, the MetaOCaml web site will be continually updated with new research results and academic materials relating to MSP [11].

## Acknowledgments

Stephan Jorg Ellner, John Garvin, Christoph Hermann, Samah Mahmeed, and Kedar Swadi read and gave valuable comments on this tutorial. Thanks are also due to the reviewers, whose detailed comments have greatly improved this work, as well as to the four editors of this volume, who have done an extraordinary job organizing the Dagstuhl meeting and organizing the preparation of this volume.

## References

1. Alan Bawden. Quasiquotation in LISP. In O. Danvy, editor, *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–99, San Antonio, 1999. University of Aarhus, Dept. of Computer Science. Invited talk.
2. Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
3. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.
4. Krzysztof Czarnecki<sup>1</sup>, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In this volume.

5. O. Danvy. Semantics-directed compilation of non-linear patterns. Technical Report 303, Indiana University, Bloomington, Indiana, USA, 1990.
6. Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.
7. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
8. J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *1994 ACM Conference on Lisp and Functional Programming, Orlando, Florida, June 1994*, pages 227–238. New York: ACM, 1994.
9. Xavier Leroy. Objective Caml, 2000. Available from <http://caml.inria.fr/ocaml/>.
10. Complete source code for `lint`. Available online from <http://www.metaocaml.org/examples/lint.ml>, 2003.
11. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from <http://www.metaocaml.org/>, 2003.
12. The MetaML Home Page, 2000. Provides source code and documentation online at <http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html>.
13. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000, USA. Available online from <ftp://cse.ogi.edu/pub/tech-reports/README.html>.
14. Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.
15. Tim Sheard. Accomplishments and research challenges in meta-programming. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineer SIGPLAN/SIGSOFT Conference, GPCE 2002*, volume 2487 of *Lecture Notes in Computer Science*, pages 2–44. ACM, Springer, October 2002.
16. Tim Sheard, Zine El-Abidine Benaissa, and Emir Pašalić. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, 1999. USENIX.
17. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [13].
18. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Boston, 2000. ACM Press.
19. Walid Taha and Henning Makholm. Tag elimination – or – type specialisation is a type-indexed effect. In *Subtyping and Dependent Types in Programming, APPSEM Workshop*. INRIA technical report, 2000.
20. Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 257–275, 2001.
21. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.
22. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.

# DSL Implementation in MetaOCaml, Template Haskell, and C++

Krzysztof Czarnecki<sup>1</sup>, John T. O'Donnell<sup>2</sup>, Jörg Striegnitz<sup>3</sup>, and Walid Taha<sup>4</sup>

<sup>1</sup> University of Waterloo, Canada

<sup>2</sup> University of Glasgow, United Kingdom

<sup>3</sup> Research Centre Jülich, Germany

<sup>4</sup> Rice University, USA

**Abstract.** A wide range of domain-specific languages (DSLs) has been implemented successfully by embedding them in general purpose languages. This paper reviews embedding, and summarizes how two alternative techniques – staged interpreters and templates – can be used to overcome the limitations of embedding. Both techniques involve a form of generative programming. The paper reviews and compares three programming languages that have special support for generative programming. Two of these languages (MetaOCaml and Template Haskell) are research languages, while the third (C++) is already in wide industrial use. The paper identifies several dimensions that can serve as a basis for comparing generative languages.

## 1 Introduction

A basic design choice when implementing a programming language is whether to build an interpreter or a compiler. An interpreter realizes the actions of the program by stepping through the source program. A compiler consists of a translator and a runtime system. The translator maps the source program to a program in an existing language, while the runtime environment provides the primitives needed for the resulting program to be executable. Translation can be to a lower-level language – as is the case in traditional compilers – or to a high-level language for which we already have an implementation.

An interesting special case occurs when no translation is needed. This means that the new language is both syntactically and semantically a subset of an existing host language. In this case, all we need is to implement the runtime system as a library in the host language. This approach of *embedded languages* has recently gained significant popularity in the functional programming community [19]. Functional languages make it possible to implement DSLs that are more sophisticated than is possible with traditional languages. For example, lambda abstractions can be used conveniently to allow DSLs with binding constructs, and nontrivial type systems for the DSLs can be encoded within the sophisticated type systems provided by functional languages. From the DSL implementer's point of view, the benefits of this approach include reusing the parser, type checker, and the compiler of an existing language to implement a

new one. Examples of embedded DSLs include parsing [32, 22, 37], pretty printing [21], graphics [12, 11], functional reactive programming [20], computer music [18], robotics [40], graphical user interfaces [4], and hardware description languages [1, 31, 36]. While this approach works surprisingly well for a wide range of applications, embedding may not be appropriate if there is

- *Mismatch in concrete syntax.* A prerequisite for embedding is that the syntax for the new language be a subset of the syntax for the host language. This excludes many stylistic design choices for the new language, including potentially useful forms of syntactic sugar and notational conventions. Furthermore, as DSLs become easier to implement, we can expect an increase in the use of graphical notations and languages, which often do not use the same syntax as potential host languages.
- *Mismatch in semantics.* The embedding approach requires that the semantics for the DSL and host languages coincide for the common subset. For example, two languages may share the same syntax but not the same semantics if one is call-by-value and the other is call-by-name. A more domain-specific example arises in the Hydra language [36], which uses the same syntax as Haskell, but which has a different semantics.

Even when these problems can be avoided, the resulting implementation may be lacking in a number of ways:

- *Domain-specific type-systems.* Often DSLs are designed to have limited expressiveness. This is generally hard to achieve with embedding because the DSL inherits the type system of the host language. For instance, consider a host language with subtyping and dynamic dispatch and a DSL for which we want just static method binding.
- *Error messages.* Similarly, when an embedded program fails to type check in the host language, the error messages are expressed in terms that are often incomprehensible (and arguably irrelevant) to the DSL user.
- *Debugging support.* Trying to use the debugger for the existing language is not always useful, as it often exposes details of the runtime system for the host language which are of no interest to the DSL user.
- *Domain-specific optimizations.* Embedding does not lend itself naturally to realizing optimizations that are valid only for the domain, or that simply do not happen to be provided by the existing language implementation. This situation is further aggravated by the fact that the official definitions of many popular host languages do not specify what optimizations the compiler must perform.

**Contributions and organization of the rest of this paper.** While there have been a number of promising proposals for addressing each of these points (c.f. [39] for typing, [10, 8, 58] for optimizations), all these approaches use program manipulation and generation, which can involve substantially more effort, care, and expertise on the part of the DSL implementer [47]. Because of the potential for specialized programming language support to help in controlling

the complexity of generative programs, we compare three languages in terms of what they offer to support these approaches. The first two are the research languages MetaOCaml [2, 33] and Template Haskell [44], presented in Sections 2 and 3 respectively. The third is the C++ language (Section 4), which is already in industrial use for generative programming, although it was not designed with that purpose in mind. Throughout the paper, an attempt is made to introduce only the essential terminology for each of the languages, and to introduce terminology useful for comparing and contrasting these languages. Section 5 presents a comparative analysis of the three languages. The goal of this analysis is to identify and highlight the key differences between the three languages and the design philosophies they represent. Section 6 concludes.

## 2 MetaOCaml

MetaOCaml is a multi-stage extension of the OCaml programming language [2, 33]. Multi-stage programming languages [51, 47, 52] provide a small set of constructs for the construction, combination, and execution of program fragments. The key novelty in multi-stage languages is that they can have static type systems that guarantee *a priori* that all programs generated using these constructs will be well-typed. The basics of programming in MetaOCaml can be illustrated with the following declarations:

```
let rec power n x = (* int -> .<int>. -> .<int>. *)
  if n=0 then .<1>. else .<~x * ~(power (n-1) x)>.
let power3 = (* int -> int *)
  .! .<fun x -> ~(power 3 .<x>.)>.
```

Ignoring the code type constructor `.<t>.` and the three staging annotations brackets `.<e>.`, escapes `~e` and run `!`, the above code is a standard definition of a function that computes  $x^n$ , which is then used to define the specialized function  $x^3$ . Without staging, the last step just produces a function that invokes the power function every time it gets a value for  $x$ . The effect of staging is best understood by starting at the end of the example. Whereas a term `fun x -> e x` is a value, an annotated term `.<fun x -> ~(e .<x>.)>.` is not. Brackets indicate that we are constructing a future stage computation, and an escape indicates that we must perform an immediate computation *while* building the bracketed computation. The application `e .<x>.` has to be performed even though  $x$  is still an uninstantiated *symbol*. In the `power` example, `power 3 .<x>.` is performed immediately, once and for all, and not repeated every time we have a new value for  $x$ . In the body of the definition of the `power` function, the recursive application of `power` is also escaped to make sure that they are performed immediately. The run `!` on the last line invokes the compiler on the generated code fragment, and incorporates the result of compilation into the runtime system.

The basic approach to implementing DSLs in MetaOCaml is the *staged interpreter approach* [13, 26, 25, 43]. First, an interpreter is implemented and tested for the DSL. Then, the three staging constructs are used to produce an implementation that performs the traversal of the DSL program in a stage earlier



than the execution of the essence of the program. Implementations derived in this manner can be as simple as an interpretive implementation, and at the same time have the performance of a compiled implementation.

## 2.1 A QBF Interpreter

We consider a simplistic DSL motivated by the logic of quantified boolean formulae (QBF). The syntax of such a language can be represented in OCaml as follows:

```
type bexp = True                               (* T *)
          | False                             (* F *)
          | And of bexp * bexp                (* T ^ F *)
          | Or of bexp * bexp                 (* T v T *)
          | Not of bexp                       (* not T *)
          | Implies of bexp * bexp            (* F => T *)
          | Forall of string * bexp           (* forall x. x and not x*)
          | Var of string                     (* x *)
```

Formulae such as  $\forall p. T \Rightarrow p$  can be represented using this datatype by the value `Forall ("p", Implies(True, Var "p"))`. Implementing this DSL would involve implementing an interpreter that checks the validity of the formula. Such a function is implemented concisely by the following (Meta)OCaml code:

```
exception VarNotFound;;

let env0 x = raise VarNotFound

let ext env x v = fun y -> if y=x then v else env y

let rec eval b env =
  match b with
  | True -> true
  | False -> false
  | And (b1,b2) -> (eval b1 env) && (eval b2 env)
  | Or (b1,b2) -> (eval b1 env) || (eval b2 env)
  | Not b1 -> not (eval b1 env)
  | Implies (b1,b2) -> eval (Or(b2,And(Not(b2),Not(b1)))) env
  | Forall (x,b1) ->
    let trywith bv = (eval b1 (ext env x bv))
    in (trywith true) && (trywith false)
  | Var x -> env x
```

The first line declares an exception that may be raised (and caught) by the code to follow. We implement an environment as a function that takes a name and either returns a corresponding value or raises an exception if that value is not found. The initial environment `env0` always raises the exception because it contains no proper bindings. We add proper bindings to an environment using the function `ext`, which takes an environment `env`, a name `x`, and a value `v`. It returns a new environment that is identical to `env`, except that it returns `v`

if we look up the name `x`. The evaluation function itself takes a formula `b` and environment `env` and returns a boolean that represents the truth of the formula.

## 2.2 A Staged QBF Interpreter

An interpreter such as the one above is easy to write, but has the undesirable property that it must repeatedly traverse the syntax of the DSL program while evaluating it. For the interpreter above, when a formula contains quantifiers, sub-terms will be evaluated repeatedly, and traversing them multiple times can have a significant effect on the time and space needed to complete the computation. Staging can allow us to separate the two distinct stages of computation: traversing the term and evaluating it. The above interpreter can be staged by adding brackets `.<...>.` and escapes `.~...` as follows:

```
let rec eval' b env =
  match b with
  | True -> .<true>.
  | False -> .<false>.
  | And (b1,b2) -> .< .~(eval' b1 env) && .~(eval' b2 env) >.
  | Or (b1,b2) -> .< .~(eval' b1 env) || .~(eval' b2 env) >.
  | Not b1 -> .< not .~(eval' b1 env) >.
  | Implies (b1,b2) -> .< .~(eval' (Or(b2,And(Not(b2),Not(b1)))) env) >.
  | Forall (x,b1) ->
    .< let trywith bv = .~(eval' b1 (ext env x .<bv>..))
      in (trywith true) && (trywith false) >.
  | Var x -> env x
```

Whereas `eval` interleaves the traversal of the term with performing the computation, `eval'` traverses the term exactly once, and produces a program that does the work required to evaluate it. This is illustrated by the following session in the MetaOCaml interactive bytecode interpreter:

```
# let a = eval' (Forall ("p", Implies(True, Var "p"))) env0;;
a : bool code =
.<let trywith = fun bv -> (bv || ((not bv) && (not true)))
  in ((trywith true) && (trywith false))>.
```

Note that the resulting computation is using only native OCaml operations on booleans, and there are no residual representations of the DSL program that was parsed.

MetaOCaml allows us to execute generated code fragments by invoking the compiler on each code fragment, and then incorporating the result into the run-time environment. This is done concisely using the `run` construct `.!:`

```
# .! a;;
- : bool = false
```

## 2.3 Domain-Specific Analysis and Optimizations

Because functional programming languages have concise notation for representing parse trees (algebraic datatypes), domain-specific optimizations can be de-

defined concisely in MetaOCaml. Pattern matching is particularly convenient when we want to implement such optimizations:

```
let rec optimize e =
  match e with
  ...
  | Or (e, False) -> optimize e
  | And (e, True) -> optimize e
  ...
```

Note that here we are pattern matching on the datatype used to represent the DSL programs, not on the MetaOCaml code type (which is the return type of the staged interpreter above). Pattern matching on the code type is not encouraged<sup>1</sup>, primarily because it is currently not known how to allow it without either allowing ill-typed code to be generated or to make substantial changes to the MetaOCaml type system [49]. This way the programmer is sure that once a particular code fragment is generated, the meaning of this code fragment will not change during the course of a computation.

## 2.4 The User's View

Staged interpreters can be provided as libraries that have the same interface as a traditional interpreter. For example, we can redefine `eval` as:

```
# let eval a env = .! (eval' a env);;
```

This function has the same signature as the one above, but different performance characteristics. Because MetaOCaml is an extension of a general purpose programming language (OCaml), implementing parsers and loaders is not problematic, and it is easy to see how we can directly execute small DSL programs written in concrete syntax, and load bigger ones from a file:

```
# eval (parse "forall x. x and not x");;
# eval (parse (load "myTheorem.dsl"));
```

## 3 Template Haskell

Template Haskell is an extension of Haskell that supports compile-time preprocessing of Haskell source programs [9, 44]. The goal of Template Haskell is to enable the Haskell programmer to define new language features without having to modify the compilers [28, 29, 16].

To illustrate some of the constructs of Template Haskell, we present an analog of the power example above. The `expand_power` function builds an expression that multiplies `n` copies of the expression `x` together, and `mk_power` yields a function that computes  $x^n$  for any integer  $x$ .

<sup>1</sup> Using coercions it is possible to expose the underlying representation of the MetaOCaml code type, which is the same as the datatype for parse trees used in the OCaml compiler.

```

module A where
import Language.Haskell.TH.Syntax

expand_power :: Int -> Q Exp -> Q Exp
expand_power n x =
  if n==0
  then [| 1 |]
  else [| $x * $(expand_power (n-1) x ) |]

mk_power :: Int -> Q Exp
mk_power n = [| \x -> $(expand_power n [| x |]) |]

```

A specialized function `power3` is now defined using `$` to splice in the code produced at compile time by `mk_power 3`.

```

module Main where
import Language.Haskell.TH.Syntax
import A

power3 :: Int -> Int
power3 = $(mk_power 3)

```

In this example, quotations `[|e|]` and splice `$e` are analogous to the brackets and escape of MetaOCaml. However, there are three key differences. First, in place of MetaOCaml's parameterized type constructor `.<t>`, the type for quoted values in Template Haskell is always `Q Exp`. Second, in the definition of `module Main`, the splice occurs without any explicit surrounding quotations. To compare this with MetaOCaml, the module (apart from the `import` directive) could be interpreted as having implicit quotations<sup>2</sup>. Third, there is no explicit run construct.

The staged interpreter approach can be used in Template Haskell. A Template Haskell version of the QBF Interpreter example would be similar to the MetaOCaml version. Programs can be staged using quasi-quotes and splices. But the primary approach to implementing DSLs in Template Haskell is a variation of the embedding approach. In particular, Template Haskell allows the programming to alter the semantics of a program by transforming it into a different program before it reaches the compiler. This is possible because Template Haskell allows the inspection of quoted values.

### 3.1 Building and Inspecting Abstract Syntax Trees

Template Haskell provides an algebraic data type for representing Haskell code as an abstract syntax tree. A part of the datatype is the following:

```

data Lit = ... | Integer Integer | ...
data Exp = ... | Var String | Lit Lit
          | App Exp Exp | ... | Tup [Exp] | ...

```

---

<sup>2</sup> This analogy has in fact been formalized for macro languages [14, 50].

```

data Pat = PLit Lit | Pvar String | Ptup [Pat] | ...
data Dec = Fun String [Clause] | Val Pat RightHandSide [Dec] | ...
data Clause = Clause [Pat] RightHandSide [Dec]
data RightHandSide = Guarded [(Exp,Exp)] | Normal Exp

```

In addition to quotations, trees can be built explicitly using the constructors shown above. Template Haskell also provides a set of monadic operations<sup>3</sup> to construct syntax trees, which offer better support for variable renaming and input/output.

Alternatively, the abstract syntax tree may be defined using a quoted expression, which is more readable than explicit construction. The entire contents of a module may be quoted, in order to produce a list of abstract syntax trees comprising the entire contents of the module. An example of a quoted definition (which represents a program in the Hydra DSL) is:

```

[| circ x y = (a,b)
   where a = inv x
         b = xor2 a y |]

```

The value of the quoted expression is a monadic computation that, when performed, produces the abstract syntax tree:

```

Fun "circ"
  [Clause
    [Pvar "x'0", Pvar "y'1"]
    (Normal (Tup [Var "a'2", Var "b'3"])))
    [Val (Pvar "a'2") (Normal (App (Var "Signal:inv") (Var "x'0")))] [],
    Val (Pvar "b'3") (Normal (App (App (Var "Signal:xor2")
                                      (Var "a'2")) (Var "y'1")))] []]]

```

This abstract syntax tree consists of one clause corresponding to the single defining equation. Within the clause, there are three pieces of information: the list of patterns, the expression to be returned, and a list of local definitions for **a** and **b**. The variables have been renamed automatically in order to prevent accidental name capture; for example **x** in the original source has become **x'0** in the abstract syntax tree. The numbers attached to variable names are maintained in a monad, and the DSL programmer can create new names using a monadic `gensym` operation.

In general, it is easier to work with code inside `[| |]` brackets; this is more concise and the automatic renaming of bound variables helps to maintain correct lexical scoping. However, the ability to define abstract syntax trees directly offers more flexibility, at the cost of added effort by the DSL implementor.

Yet another way to produce an abstract syntax tree is to reify an ordinary Haskell definition. For example, suppose that an ordinary Haskell definition `f x = ...` appears in a program. Although this user code contains no quotations or

---

<sup>3</sup> In Haskell, computations are implemented using an abstract type constructor called a monad [34, 41].

splices, the DSL implementation can obtain its abstract syntax tree: `ftree = reifyDecl f`. Values and types can be reified, as well as operator fixities (needed for compile time code to understand expressions that contain infix operators) and code locations (which enables the compile time code to produce error messages that specify where in the source file a DSL error occurs).

Once a piece of Haskell code has been represented as an abstract syntax tree, it can be analyzed or transformed using standard pattern matching in Haskell. This makes it possible to transform the code into any Haskell program that we wish to use as the DSL semantics for this expression.

### 3.2 The User's View

As with MetaOCaml, the staged interpreter can be made available to the user as a library function, along with a parser and a loader. Such functions can only be used at compile time in Template Haskell. It is worth noting, however, that Template Haskell does allow compile-time IO, which is necessary if we want to allow loading DSL programs from files.

Additionally, Template Haskell's quotations make it possible to use Haskell syntax to represent a DSL program. Template Haskell also makes it possible to quote declarations (and not just expressions) using the `[d| ... |]` operator:

```
code = [d|
  v1 = [1,2,3]
  v2 = [1,2,3]
  v3 = [1,2,3]
  r = v1 'add' (v2 'mul' v3)
|]
```

Then, another module could apply a staged interpreter to this code in order to transform the original code into an optimized version and splice in the result.

Template Haskell also allows templates to construct declarations that introduce new names. For example, suppose we wish to provide a set of specialized power functions, `power2`, `power3`, up to some maximum such as `power8`. We can write a Template Haskell function that generates these functions, using the `mk_power` defined above, along with the names, and then splices them into the program:

```
$(generate_power_functions 8)
```

## 4 C++ Templates

Originally, templates were intended to support the development of generic containers and algorithms. Driven by the practical needs of library implementers, the C++ template mechanism became very elaborate. By chance rather than by design, it now constitutes a Turing-complete, functional sub-language of C++ that is interpreted by the C++ compiler [54, 57]. Consequently, a C++ program

may contain both static code, which is evaluated at compile time, and dynamic code, which is compiled and later executed at runtime. Static computations are encoded in the C++ type system, and writing static programs is usually referred to as Template Metaprogramming [57].

Before giving a short overview of what static code looks like, we recapitulate some C++ template basics.

## 4.1 Template Basics

C++ supports two kinds of templates: class templates and function templates. The basic use of class templates in C++ is to define generic components. For example, we can define a generic vector as a class template with element type and size as parameters:

```
template <class T, int size> class Vector
{public:
    T& operator[](int i);
    //...
    T data[size];
};
```

A template is *instantiated* in a `typedef` statement as follows:

```
typedef Vector<float,10> MyVector;
```

Here, the type parameter `T` in the original definition is simply replaced by `float` and `size` by `10`. Now we can use `MyVector` as a regular C++ class:

```
MyVector v;
v[3] = 2.2;
```

A template can be *specialized* by providing a definition for a specialized implementation as follows:

```
template <int size>
class Vector<bool, size>
{ //...
    // use just enough memory for 'size' bits
    char data[(sizeof(char)+size-1)/sizeof(char)];
};
```

Such specializations are useful because one general implementation of a template might not be efficient enough for some argument values. Now, whenever we instantiate `Vector` with `bool` as its first parameter, the latter *specialized* implementation will be used.

Function templates are parameterized functions. A function template to swap two values may be implemented like this:

```
template <class T>
void swap (T& a, T& b)
{ const T c = a; a = b; b = c; }
```

The function template `swap` can be called as if it were an ordinary C++ function. The C++ compiler will automatically infer and substitute the appropriate parameter type for `T`:

```
int a = 5, b = 9;
swap(a,b);
```

## 4.2 Template Metaprogramming

Template Metaprogramming adds two rather unusual views of class templates: class templates as data constructors of a datatype and class templates as functions. To introduce these views, we will compare them with the corresponding Haskell definitions.

Here is an example of C++ template program defining a list, followed by the corresponding Haskell code.

**C++:**

```
struct Nil {};
template <int H, class T>
struct Cons {};
```

`typedef`

```
Cons<1,Cons<2, Nil> > list;
```

**Haskell:**

```
data List = Nil | Cons Int List
```

```
list = Cons 1 (Cons 2 Nil)
```

The C++ definitions for `Nil` and `Cons` have different dynamic and static semantics. Dynamically, `Nil` is the name of a class, and `Cons` is the name of a class template. Statically, both `Nil` and `Cons` can be viewed as data constructors corresponding to the Haskell declaration on the right. All such constructors can be viewed as belonging to a single “extensible datatype”, which such definitions extend.

The C++ template instantiation mechanism [23] provides the semantics for compile-time computations. Instantiating a class template corresponds to applying a function that computes a class. In contrast, class template *specialization* allows us to provide different template implementations for different argument values, which serves as a vehicle to support pattern matching, as common in functional programming languages. Next, we compare how functions are implemented at the C++ compile time and in Haskell:

**C++:**

```
template <class List>
struct Len;
template <>
struct Len<Nil> {
    enum { RET = 0 }; };

```

```
template <int H,class T>
struct Len<Cons<H,T> > {
    enum { RET = 1 + Len<T>::RET }; };

```

**Haskell:**

```
len :: List -> Int;
```

```
len Nil = 0
```

```
len (Cons h t) = 1 + (len t)
```



The template specializations `Len<Nil>` and `Len<Cons<H,T> >` handle the empty and non-empty list cases, respectively. In contrast to Haskell or ML, the relative order of specializations in the program text is insignificant. The compiler selects the “best match” (which is precisely defined in the C++ standard [23]).

This calling scheme makes static functions very extensible. Having an extensible datatype and extensible functions, C++ Template Metaprogramming does not suffer from the extensibility problem [27].

The `::` operator is used to access the name space of the class. For example, `Len<list>::RET` will evaluate to 2. The member being accessed could also be a computed type or a static member function. Template Metaprogramming also supports higher-order functions through so-called template template parameters.

In relation to DSL implementation, C++ templates can be used in two distinct ways: at the type level, and at the expression level. The first is comparable to Template Haskell’s mechanism to analyze and build new declarations. The second is comparable to the staged interpreters approach, except that no parser has to be implemented.

### 4.3 Type-Driven DSLs

Template Metaprogramming allows us to write *configuration generators* [5]. Typically, such generators must check parameters, compute default values, and perform any computations necessary for assembling generic components. Syntactically, type-driven DSLs are subsets of C++ type expressions. To give a concrete example of such a type-driven DSL, consider a generator for different implementations of matrix types. The DSL program (which results in invoking the generator) can be written as follows:

```
typedef MatrixGen<ElemType<float>,Optim<space>,Shape<u_triang> >::RET M1;
typedef MatrixGen<Shape<l_triang>,Optim<speed> >::RET M2;
```

`MatrixGen` takes a list of properties, checks their validity, completes the list of properties by computing defaults for unspecified properties, and computes the appropriate composition of elementary generic components such as containers, shape adapters and bounds checkers. This configuration generator uses a specific style for specifying properties that simulates keyword parameters: for example, when we write `ElemType<float>` we have a parameter `float` wrapped in the template `ElemType` to indicate its name [7]. Configuration generators can compose mixins (that is, class template whose superclasses are still parameters) and thus generate whole class hierarchies [5]. They can also compose mixin layers [45], which corresponds to generating object-oriented frameworks.

### 4.4 Expression-Driven DSLs

A popular technique for implementing expression-level embedded DSLs is expression templates [56, 17]. To illustrate this approach, we consider a simple DSL of vector expressions such as `(a+b)*c`. This DSL could easily be embedded by overloading the `+` and `*` operators to perform the desired computations.

However, assume that our DSL should have parallel execution semantics, e.g., by using OpenMP[3]. In order to achieve this, we need to transform the expression into code like the following:

```
// Tell the OpenMP compiler to parallelize the following loop.
#pragma OMP parallel for
for (int i=0 ; i< vectorSize; ++i)
    // perform componentwise evaluation of a vector
    // expression, e.g., (a[i] + b[i]) * c[i])
```

This code cannot be produced by simple operator overloading since each operator would contribute an additional loop. However, expression templates and template metaprograms can be used to generate the desired code. The gist of expression templates is to overload operators to return an object that reflects the structure of the expression in its type. This has the effect of turning the expression into a parse tree at compile-time. Here is how we can reflect the parse tree of the expression: the leaves of the parse tree will be vectors and we will use BNode to represent non-leaf nodes:

```
struct OpPlus {};
struct OpMult {};

template <class OP,class Left,class Right> struct BNode
{ BNode(const Op& o,const Left& l,const Right& r);
  OP    Op;
  Left  left;
  Right right;
};
```

For example, the type of the expression  $(a+b)*c$ , where all variables are of type `Vector<int, 10>`, would be:

```
BNode< OpMult,
      BNode< OpPlus,
            Vector<int, 10>,
            Vector<int, 10> >,
      Vector<int, 10> >
```

To create such an object, we overload all the operators used in our DSL for all combinations of operand types (`Vector/Vector`, `Vector/BNode`, `BNode/Vector`, and `BNode/BNode`):

```
//implementation for Vector+Vector; others analogous
template <class V,int size>
BNode<OpPlus,Vector<V,size>,Vector<V,size> >
operator+(const Vector<V,size>& l,const Vector<V,size>& r) {
    return BNode<OpPlus,Vector<V,size>,Vector<V,size> >(OpPlus(), l, r );
}
```

Note that code inspection in C++ is limited. For a type-driven DSL all information is embedded in the type. For an expression-driven DSL, this is not the case. For instance, we can only inspect the type of a vector-argument; its address in memory remains unknown at the static level. Inspecting arbitrary C++ code is possible to a very limited extent. For instance, different programming idioms (such as traits members, classes, and templates [35, 30]) have been developed to allow testing a wide range of properties of types. Examples of such properties include whether a type is a pointer type or if type A is derived from B. Many other properties (such as the names of variables, functions and classes; and member functions of a class) must be hard-coded as traits.

Next, we define `Eval`, which is a transformation that takes a parse tree and generates code to evaluate it. For instance, while visiting a leaf, we generate a function that evaluates a vector at a certain position `i`:

```
template <class T,int size> struct Eval< Vector<T,size> >
{ static inline T evalAt(const Vector<T,size>& v, int i)
  { return v[i]; }
};
```

For a binary node we first generate two functions to evaluate the values of the siblings. Once these functions have been generated (i.e, the corresponding templates were instantiated), we generate calls to those functions:

```
template <class L,class R> struct Eval< BNode<OpPlus,L,R> >
{ static inline T evalAt(const BNode<OpPlus,L,R>& b,int i)
  { return Eval<L>::evalAt  // generate function
    (b.left,i)             // generate call to generated code
    + // Operation to perform
    Eval<R>::evalAt  // generate function
    (b.right,i); }        // generate call to generated code
};
```

Since all generated functions are eligible candidates for inlining and are tagged with the `inline` keyword, a compiler should be able to remove any call overhead so that the result is the same as if we spliced in code immediately.

Invoking the `Eval` transformation is somewhat tedious because we need to pass the complete parse tree to it. Since type parameters to function templates are inferred automatically, a function template solves this problem quite conveniently:

```
template <class T>
T inline eval(const T& expression,int i) {
  return Eval<T>::evalAt(expression, i);
}
```

When we call `eval((a+b)*c,1)`, the compiler instantiates this function template, which will automatically bind `T` to the type of `(a+b)*c`. The compiler will then expand `Eval<T>::evalAt(e,i)` recursively into efficient code to evaluate `expression` at index `i`.

The machinery just discussed delays execution of the expression (staging), gives access to the structure of programs (intensional analysis) and provides a means to generate code that evaluates the expression (code splicing). In fact, we can view `evalAt` function as an example of a staged interpreter in C++.

The remaining task is to make the `Vector` class aware of the new evaluation scheme. That is, we have to define how to initialize a vector from an expression and how to assign an expression to it:

```
template <class A,class Expression,int size>
Vector<A,size>& Vector<A,size>::operator=(const Expression& rhs) {
#pragma omp parallel for
    for (int i=0 ; i<size ; ++i)
        { data[i] = eval( rhs, i ); }
    return *this;
};
```

The code generated for the assignment statement `d=((a+b)*c);` will be the same as the desired code given at the beginning of this section with the inner comment replaced by `d[i]=(a[i]+b[i])*c[i]`, and it will be executed in parallel.

## 5 Comparison

In this section, we present some basic dimensions for variation among the three different languages. The following table summarizes where each of the languages falls along each of these dimensions:

Dimension	MetaOCaml	Template Haskell	C++ Templates
1. Approach	Staged interp.	Templates	Templates
2. How	Quotes	Quotes & abs. syn.	Templates
3. When	Runtime	Compile-time	Compile-time
4. Reuse	Compiler	Compiler (& parser)	Compiler & parser
5. Guarantee	Well-typed	Syntax valid	Syntax valid
6. Code inspection	No	Yes	Limited
7. IO	Always	Always	Runtime
8. Homogeneous	Yes	Yes	No
9. CSP	Yes	Yes	No
10. Encapsulation	Yes	No	Yes
11. Modularity	Yes	Yes	No

The definition (and explanation) of each of these dimensions is as follows:

1. Of the DSL implementation approaches discussed in the introduction, what is the primary approach supported? To varying extents, both Template Haskell and C++ can also support the staged interpreter approach (but with a different notion of static typing). MetaOCaml staged interpreters can be translated almost mechanically into Template Haskell. Staging in C++ is not as straightforward due to its heterogeneity. Also, in C++, staged interpreters must use C++ concrete syntax (see Reuse and IO dimensions below).

2. How is code generated? MetaOCaml and Template Haskell generate code through quotes and splices. Additionally, Template Haskell allows splicing explicitly-constructed abstract syntax trees. C++ Template Metaprogramming generates code through code selection (expressed using template specialization) and inlining. MetaOCaml and C++ perform all necessary renaming automatically<sup>4</sup>. Because Template Haskell allows access to the constructors for building Haskell parse trees, it must also provide an explicit `gensym` function.
3. When is code generated? MetaOCaml allows any subcomputation to involve generation of new code. Template Haskell and C++ limit code generation to compile time. MetaOCaml is called a *multi-stage language* because it provides constructs for both the construction and execution of new code during runtime [47].
4. What part of the implementation of the host language can be reused? Each of the languages considered in this paper can be viewed as providing a means for the programmer to reuse a particular aspect of the implementation of an existing language. All three languages reuse the compiler, because they allow the compiler for the host language to be used in compiling the DSL. The staged interpreter approach in MetaOCaml requires implementing a parser for the DSL. Template Haskell gives the choice of implementing a parser or reusing the Haskell parser in the context of implementing embedded languages. C++ Template Metaprogramming requires reusing the C++ parser.
5. What does the type system guarantee about *generated programs*? When we represent programs as datatypes in a statically typed language, the only runtime guarantee that the static type system provides is that a value of this datatype is a syntactically well-formed program. In MetaOCaml, a stronger guarantee is possible, namely, that any generated program is well-typed. But it is useful to note that if at generation-time, code-to-code coercions are used to circumvent the MetaOCaml type system, this guarantee is lost *but type safety is not lost*. In particular, because MetaOCaml also type checks all code before compiling it (as do the other two languages), as long as the generated code does not contain coercions, the system is still safe. In addition, when the MetaOCaml type-checker rejects a dynamically generated program (for example because coercions were used when generating it) an OCaml exception is raised, and the DSL implementer can take appropriate provisions to catch and handle such exceptions. In the other languages, any such failure is reported directly to the user of the DSL. As an aside, note that C++ does no type checking on the templates before they are instantiated. Thus, compilation-time runtime errors can occur. Still, the result of instantiation is always syntactically valid.
6. Can code be inspected (intensional analysis)? MetaOCaml discourages code inspection, Template Haskell allows all code inspection, and code inspection in C++ lies between these extremes. First, only a limited form of code in-

---

<sup>4</sup> For a subset of MetaOCaml, it has been shown that all necessary renaming is done [2].

spection can be achieved through expression templates and traits. Second, the scope of transformations in the expression-driven DSLs is limited to a single C++ expression, i.e., no global optimizations are possible. However, a sequence of DSL statements can be represented as a single C++ expression by using the overloaded comma operator as a separator, allowing the transformation of several DSL statements at once [24, 46]. For staged interpreters, the ability to inspect generated code is not necessary in order to implement domain-specific optimizations, since those optimizations can be defined directly on the datatype representing the syntax of DSL programs.

7. When can IO be performed? For the embedded DSL approach, IO is essential for reporting errors to the user during analysis and type checking. Template Haskell supports IO at expansion time. Besides error reporting, IO also has other applications such as inspecting the code being generated for debugging purposes (which is not possible in C++), reading additional specifications from some file (e.g., an XML configuration file), or generating code in other languages (e.g., C for interfacing to some native libraries). C++ does not allow any IO during template expansion; errors can be reported only using a rather crude programming trick by enforcing a compiler error reporting an identifier with the error cause as its name [6, p. 669]. Note that a staged interpreter in MetaOCaml is just a program, and so it has access to all IO available in the OCaml language.
8. Are both generator and generated languages the same? A generative language is said to be *homogeneous* when both are identical, and *heterogeneous* otherwise [47]. Homogeneity makes the process of staging a program (including an interpreter) a matter of adding staging annotations. It also makes it easy to cut and paste terms in and out of quotations, and only one set of programming skills is required for learning both the generating and the generated language. In MetaOCaml, only expressions can be quoted. But expressions in MetaOCaml can contain declarations, modules, new classes, etc. Template Haskell can generate expressions and declarations of types, functions, values, classes, and instances. Only expressions and declarations can be spliced in (but not, e.g., patterns). Note, however, that Template Haskell does not allow generating code containing quotes or splices (unlike MetaOCaml). C++ can generate classes (and class hierarchies; see Section 4.3), functions, expressions (using expression templates; see Section 4.4), and templates (as members). Functions can be generated using function templates, e.g., `power<3>(int 4)` could be a call to a function template taking the exponent as a static template parameter and generating code with an unrolled loop [6, p. 495].
9. Is cross-stage persistence (CSP) [51] supported? CSP makes it possible to embed values that are outside quotations directly into quoted terms (for example, we can write `.< reverse (append 1.1 1.2) >.` even when `reverse` and `append` refer to actual functions defined outside quotations). This means only that the generated code can refer dynamically to the generator environment, and not necessarily that the languages used for both are the same (homogeneity).

10. Is generativity encapsulated? In MetaOCaml, the `run` construct makes it possible for a library internally to generate and execute code. It is also the case that if all generated code is well-typed, then compilation cannot fail, and the user of the library never needs to deal with obscure error messages. Thus, the user of the library does not need to know that generativity is being used, and we can therefore say that it is well *encapsulated*. In C++, templates can be used to provide libraries that appear to the user as being no different from a non-generative implementation. In Template Haskell, the use of templates in libraries can be encapsulated if these templates do not use parameters that the library user needs to provide. Otherwise, an explicit splice must be used in the application code.
11. Is there support for separate compilation of modules?

## 6 Discussion and Concluding Remarks

When applicable, embedding can be the easiest way to implement a DSL. Because we may wish to write different applications in different DSLs, embedding may also help interoperability. But this approach may not be applicable if there is a mismatch in syntax, semantics, optimizations, debugging, or error reporting. Languages such as MetaOCaml, Template Haskell and C++ provide alternative, more flexible approaches to implementing DSLs. This paper has presented a brief introduction to these three languages, along with a comparative analysis of them.

The large number of practical DSL implementations using generative programming in C++ shows that generative programming support is useful – and sometimes necessary – in a mainstream language. This support is especially important for library programmers. The success of generative programming in C++ is remarkable, given that C++ was not designed for DSL implementation. C++ Template Metaprogramming suffers from a number of limitations, including portability problems due to compiler limitations (although this has significantly improved in the last few years), lack of debugging support or IO during template instantiation, long compilation times, long and incomprehensible errors, poor readability of the code, and poor error reporting. Nevertheless, these limitations have not prevented people from creating useful embedded DSLs that would have been impossible otherwise (such as ones for vector computations [55] as well as numerous others that followed). Collections of library primitives to make generative programming in C++ easier are now available [7, p. 433-451], [15].

MetaOCaml and Template Haskell are designed specifically to support generative programming: their capabilities are currently being explored and the languages as well as implementations are still evolving. A collection of small-scale projects developed by Rice University students (including implementations of SSH, parsers, a subset of XSMML, and small programming languages) is available online [42]. Experience with Template Haskell is also currently being documented [28, 29, 16], and the implementations of several DSLs in Template Haskell are currently under way, including Hydra’2003 and a parallel skeleton project by Kevin Hammond and colleagues.

Experience with these languages is revealing new challenges. Proper domain-specific error reporting is both a challenge and an opportunity: if a library told you in an understandable way that you are using it ineffectively or incorrectly, it could be much more friendly to end users. However, all three of the languages compared here lack comprehensive facilities for debugging implementations of embedded DSLs. Providing proper debugging support for the user of a DSL is largely unexplored, too. Foundational problems that deserve further exploration in program generation include more expressive type systems, type safe ways for inspecting quoted expressions, and techniques for minimizing generation and compilation times. First generation of generative languages such as the ones discussed in this paper will help us gain practical experience with building real applications. No doubt, this experience will feed back to improve our understanding of how a programming language can provide better support for generative programming.

## Acknowledgments

Anonymous reviewers, Van Bui, Simon Helsen, Roumen Kaiabachev, and Kedar Swadi provided valuable comments on drafts of this paper.

## References

1. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999.
2. Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
3. Rohit Chandra, Leonardo Dagum, and Dave Kohr. *Parallel Programming in OpenMP++*. Morgan Kaufmann, 2000.
4. Antony Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of Symposium on Practical Aspects of Declarative Languages*. ACM, 2001.
5. K. Czarnecki and U. W. Eisenecker. Synthesizing objects. In *Proceedings of ECOOP’99*, LNCS 1628, pages 18–42. Springer-Verlag, 1999.
6. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
7. K. Czarnecki and U. W. Eisenecker. Named parameters for configuration generators. <http://www.generative-programming.org/namedparams/>, 2000.
8. K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries (extended abstract). In M. Jazayeri, D. Musser, and R. Loos, editors, *Generic Programming. Proceedings*, volume 1766 of *LNCS*, pages 25–39. Springer-Verlag, 2000.
9. Simon Peyton Jones (ed.). Haskell 98 language and libraries. *Journal of Functional Programming*, 13(1):1–255, January 2003.
10. Conal Elliott, Sigbjørn Finne, and Oege de Moore. Compiling embedded languages. In [48], pages 9–27, 2000.



11. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
12. Sigbjørn Finne and Simon L. Peyton Jones. Pictures: A simple structured graphics model. In *Proceedings of Glasgow Functional Programming Workshop*, July 1995.
13. Yhoshihiko Futamura. Partial evaluation of computation: An approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
14. Steven Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *the International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001. ACM.
15. Aleksey Gurtovoy. Boost MPL Library (Template metaprogramming framework). <http://www.boost.org/libs/mpl/doc/>.
16. K. Hammond, R. Loogen, and J. Berhold. Automatic Skeletons in Template Haskell. In *Proceedings of 2003 Workshop on High Level Parallel Programming, Paris, France*, June 2003.
17. Scott Haney, James Crotinger, Steve Karmesin, and Stephen Smith. Pete: The portable expression template engine. *Dr. Dobbs Journal*, October 1999.
18. P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.
19. Paul Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.
20. Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.
21. J. Hughes. Pretty-printing: an exercise in functional programming. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction; Second International Conference; Proceedings*, pages 11–13, Berlin, Germany, 1993. Springer-Verlag.
22. G. Hutton. Combinator parsing. *Journal of Functional Programming*, 1993.
23. ISO/IEC. Programming languages – C++. ISO/IEC 14882 Standard, October 2003.
24. Jaakko Järvi and Gary Powell. The lambda library: Lambda abstraction in C++. In *Second Workshop on C++ Template Programming, Tampa Bay, Florida, USA*, October 2001.
25. Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.
26. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
27. Shriram Krishnamurti, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In Eric Jul, editor, *European Conference in Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer Verlag, 1998.
28. Ian Lynagh. Template Haskell: A report from the field. <http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/>, May 2003.
29. Ian Lynagh. Unrolling and simplifying expressions with Template Haskell. <http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/>, May 2003.
30. John Maddock and Steve Cleary et al. Boost type traits library. [http://www.boost.org/libs/type\\_traits/](http://www.boost.org/libs/type_traits/).

31. John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society Press, 1998.
32. M. Mauny. Parsers and printers as stream destructors embedded in functional languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 360–370. ACM/IFIP, 1989.
33. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from <http://www.metaocaml.org/>, 2003.
34. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.
35. N. C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5), June 1995.
36. John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings 16th International Parallel & Distributed Processing Symposium*, page 234 (abstract). IEEE Computer Society, April 2002. Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications – PDSECA.
37. Chris Okasaki. Even higher-order functions for parsing or why would anyone ever want to use a sixth-order function? *Journal of Functional Programming*, 8(2):195–199, March 1998.
38. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000, USA. Available online from <ftp://cse.ogi.edu/pub/tech-reports/README.html>.
39. Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.
40. J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conf. on Robotics and Automation*, 1999.
41. Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *the Symposium on Principles of Programming Languages (POPL '93)*. ACM, January 1993. 71–84.
42. Rice Students. Multi-stage programming course projects. <http://www.cs.rice.edu/~taha/teaching/>, 2000.
43. Tim Sheard, Zine El-Abidine Benaissa, and Emir Pašalić. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL '99)*, Austin, Texas, 1999. USENIX.
44. Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.
45. Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.
46. Jörg Striegnitz and Stephen Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 2000.
47. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [38].
48. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.

49. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Boston, 2000. ACM Press.
50. Walid Taha and Patricia Johann. Staged notational definitions. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
51. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.
52. Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.
53. Peter Thiemann. Programmable Type Systems for Domain Specific Languages. In Marco Comini and Moreno Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier, 2002.
54. Ervin Unruh. Prime number computation. Internal document, ANSI X3J16-94-0075/ISO WG21-462, 1994.
55. Todd Veldhuizen and Kumaraswamy Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobbs's Journal of Software Tools*, 21(8):38–44, August 1996.
56. Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.
57. Todd L. Veldhuizen. Template metaprograms. *C++ Report*, 7(4):36–43, 1995.
58. D. Wile. Popart: Producer of parsers and related tools. system builders' manual. Technical report, USC Information Sciences Institute, 1981.

# Program Optimization in the Domain of High-Performance Parallelism

Christian Lengauer

Fakultät für Mathematik und Informatik  
Universität Passau, D-94030 Passau, Germany  
`lengauer@fmi.uni-passau.de`

**Abstract.** I consider the problem of the domain-specific optimization of programs. I review different approaches, discuss their potential, and sketch instances of them from the practice of high-performance parallelism. Readers need not be familiar with high-performance computing.

## 1 Introduction

A program that incorporates parallelism in order to increase performance (mainly, to reduce execution time) is particularly difficult to write. Apart from the fact that its performance has to satisfy higher demands than that of a sequential program (why else spend the additional money and effort on parallelism?), its correctness is also harder to establish. Parallelism makes verification more difficult (one must prove the correctness not only of the individual processes but also of any overt or covert interactions between them) and testing less effective (parallel behaviour can, in general, not be reproduced).

Note that introducing parallelism for high-performance is a form of program optimization since, in high-performance computing, parallelism is never part of the problem specification. If one can get along without it – so much the better!

The present state of programming for high-performance parallel computers can be compared with the state of programming for sequential computers in the 1960s: to a large extent, one is concerned with driving a machine, not with solving a computational problem. One specifies, in the source program, the composition and synchronization of the parallel processes, and the communication between them. This can be viewed as assembly-level programming.

Thus, researchers in programming high-performance parallelism have been searching for commonly useful abstractions – much like researchers in sequential programming languages and their methodologies have been back in the 1960s and 1970s. Just as back then, their efforts are being resisted by the programmers in practice who feel that they need low-level control in order to tune program performance – after all, the performance is the sole justification for their existence! Just as back then, the price they pay is a lack of program structure, robustness and portability. However, one additional challenge in high-performance parallelism is the lack of a commonly accepted parallel machine model (for sequential programs we have the von Neumann model). One consequence is that, even if

the parallel program may be portable, its performance may not be but may differ wildly on different platforms<sup>1</sup>.

If we consider high-performance parallelism an application domain, it makes sense to view the programming of high-performance parallelism as a domain-specific activity. Thus, it is worth-while to assess whether techniques used in domain-specific program generation have been or can be applied, and what specific requirements this domain may have. To that end, I proceed as follows:

1. I start with a classification of approaches to domain-specific programming (Sects. 2–3).
2. I review some of the specific challenges in programming for the domain of high-performance parallelism (Sect. 4).
3. I review a variety of approaches to programming high-performance parallelism, place them in the classification of approaches to domain-specific programming and discuss their principles and limitations (Sects. 5–8). Here, I mention also all approaches to high-performance parallelism which are presented in this volume.
4. I conclude with a review of the main issues of raising the level of abstraction in programming parallelism for high performance (Sect. 9).

## 2 Domain-Specific Programming

What makes a language domain-specific is not clear cut and probably not worth worrying about too much. A debate of this issue would start with the already difficult question of what constitutes a domain. Is it a collection of users or a collection of software techniques or a collection of programs...?

For the purpose of my explorations we only need to agree that there are languages that have a large user community and those that have a much smaller user base, by comparison. Here I mean “large” in the sense of influence, manpower, money. I call a language with a large user community *general-purpose*. Probably undisputed examples are C and C++, Java and Fortran but, with this definition, I could also call the query language SQL general-purpose, which demonstrates that the term is meant in a relative sense.

A large user community can invest a lot of effort and resources in developing high-quality implementations of and programming environments for their language. A small user community has much less opportunity to do so, but may have a need for special programming features that are not provided by any programming language which other communities support. I call such features *domain-specific*. One real-world example of a language with a small user community, taken from Charles Consel’s list of domain-specific languages on the Web, is the language Devil for the specification of device driver interfaces [1].

What if the small user community prefers some widely used, general-purpose language as a base, but needs to enhance it with domain-specific constructs to

---

<sup>1</sup> On present-day computers with their memory hierarchies, instruction-level parallelism and speculation, a lack of performance portability can also be observed in sequential programs.

support its own applications? High-performance computing is such a community: it needs an efficient sequential language, enhanced with a few constructs for parallelism, synchronization and, possibly, communication. It has two options:

- It can view the combination of the two as a new, domain-specific language and provide a dedicated, special-purpose implementation and programming environment for it. However, this will put it at a disadvantage with respect to the large user community of the base language, if it is not able to muster a competitive amount of resources for the language development.
- It can attach a domain-specific extension to the base language [2]. In this case, the small community need not invest in the development of the general-purpose part of its language. Indeed, if the enhancement is crafted carefully, it may be possible to plug in a new release of the base language implementation, which has been developed independently, without too much trouble.

Let us look more closely at the latter approach for our particular purpose: program optimization.

### 3 Domain-Specific Program Optimization

Two commonly recognized benefits of domain-specific program generation are increased programming convenience and program robustness. But there is also the opportunity for increased program performance – via optimizations which cannot be applied by a general-purpose implementation. In many special-purpose domains, performance does not seem to be a major problem, but there are some in which it is. High-performance computing is an obvious one. Another is the domain of embedded systems, which strives for a minimization of chip complexity and power consumption, but which I will not discuss further here (see the contribution of Hammond and Michaelson [3] to this volume).

There are several levels at which one can generate domain-specifically optimized program parts. Here is a hierarchy of four levels; the principle and the limitations of each level are discussed further in individual separate sections:

**Sect. 5: *Precompiled Libraries.*** One can prefabricate domain-specific program pieces independently, collect them in a library, and make them available for call from a general-purpose programming language.

**Sect. 6: *Preprocessors.*** One can use a domain-specific preprocessor to translate domain-specific program pieces into a general-purpose language and let this preprocessor perform some optimizations.

**Sect. 7: *Active Libraries.*** One can equip a library module for a domain-specific compilation by a program generator, which can derive different implementations of the module, each optimized for its specific call context.

**Sect. 8: *Two Compilers.*** One can use a domain-specific compiler to translate domain-specific program pieces to optimized code, which is in the same target language to which the general-purpose compiler also translates the rest of the program. The separate pieces of target code generated by the two compilers are then linked together.

The domain specificity of the optimization introduces context dependence into the program analysis. Thus, one important feature that comes for free is that the optimization of a domain-specific program part can be customized for the context of its use. This feature is present in all four approaches, but it becomes increasingly flexible as we progress from top to bottom in the list.

## 4 The Challenges of Parallelism for High Performance

Often the judicious use of massive parallelism is necessary to achieve high performance. Unfortunately, this is not as simple as it may seem at first glance:

- The program must decompose into a sufficient number of independent parts, which can be executed in parallel. At best, one should be able to save the amount in time which one invests in additional processors. This criterion, which can not always be achieved, is called *cost optimality*; the number of sequential execution steps should equal the product of the number of parallel execution steps and the number of required processors [4]. It is easy to spend a large number of processors and still obtain only a small speedup.
- Different processors will have to exchange information (data and, maybe, also program code). This often leads to a serious loss of performance. In contemporary parallel computers, communication is orders of magnitude slower than computation. The major part of the time is spent in initiating the communication, i.e., small exchanges incur a particularly high penalty. As the number of processors increases, the relative size of the data portions exchanged decreases. Striking a profitable balance is not easy.
- In massive parallelism, it is not practical to specify the code for every processor individually. In parallelism whose degree depends on the problem size, it is not even possible to do so. Instead, one specifies one common program for all processors and selects portions for individual processors by a case analysis based on the processor number<sup>2</sup>. The processor number is queried in decisions of what to do and, more fundamentally, when to do something and how much to do. This can lead to an explosion in the number of run-time tests (e.g., of loop bounds), which can cause a severe deterioration in performance.

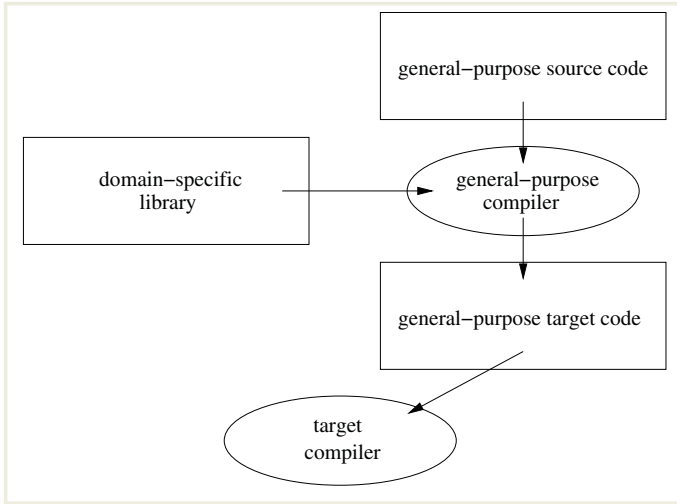
The user community of high-performance parallelism is quite large, but not large enough to be able to develop and maintain competitive language implementations in isolation [5]. Let us investigate how it is faring with the four approaches to domain-specific program optimization.

## 5 Domain-Specific Libraries

### 5.1 Principle and Limitations

The easiest, and a common way of embedding domain-specific capabilities in a general-purpose programming language is via a library of domain-specific program modules (Fig. 1). Two common forms are libraries of subprograms and, in

<sup>2</sup> This program structure is called *single-program multiple-data (SPMD)*.



**Fig. 1.** Domain-specific library

object-oriented languages, class libraries. Typically, these libraries are written in the general-purpose language from which its modules are also called.

In this approach, there is no domain-specific translation which could perform domain-specific optimizations. One way of customizing the implementation of a module for the context of its call is by asking the application programmer to pass explicitly additional, so-called structure parameters which convey context-sensitive information and which are queried in the body of the module.

The limitations of this approach are:

- Error messages generated in the library modules are often cryptic because they have been issued by a compiler or run-time system which is ignorant of the special domain.
- The caller of the module is responsible for setting the structure parameters consistently. This limits the robustness of the approach.
- The implementer of the module has to predict all contexts in which the module may be called and build a case analysis which selects the given context. This limits the flexibility of the approach.

## 5.2 Examples from High-Performance Parallelism

**MPI.** One example from high-performance parallelism is the communication library *Message-Passing Interface (MPI)* [6]. It can be implemented largely in C, but with calls to system routines for communication and synchronization. There exist MPI interfaces for Fortran and C – the popular general-purpose languages in high-performance computing<sup>3</sup>. The Fortran or C code is put on each processor in an SPMD fashion.

<sup>3</sup> There is also an MPI interface for Java, but a more natural notion of parallelism for Java is thread parallelism, as supported by JavaParty (see further on).



The routines provided by the library manage processor-to-processor communication, including the packing, buffering and unpacking of the data, and one can build a hierarchical structure of groups of processors (so-called communicators) which communicate with each other. There are over 100 routines in the first version of MPI and over 200 in the second. With such high numbers of routines it becomes difficult for programmers to orient themselves in the library and even more difficult for developers to maintain a standard.

Programming with MPI (or similar libraries like the slightly older PVM [7]) has been by far the most popular way of producing high-performance parallel programs for about a decade. As stated before, this could be likened to the sequential programming with unstructured languages or assemblers in the 1960s. Programmers have relatively close control over the machine architecture and feel they need it in order to obtain the performance they are expecting.

The library BSP [8] works at a slightly higher level of abstraction: the SPMD program is divided into a sequence of “supersteps”. Communication requests issued individually within one superstep are granted only at the superstep’s end. This alleviates the danger of deadlocks caused by circular communication requests and gives the BSP run-time system the opportunity to optimize communications.

One limitation of BSP is that there is no hierarchical structure of communicating processes (something which MPI provides with the communicator). All processors are potential partners in every communication phase.

The developers of BSP are very proud of the fact that their library consists of only 20 functions, compared with the hundreds of MPI. Still, programming with BSP is not very much simpler than programming with MPI. However, BSP has a very simple cost model – another thing its developers are proud of. The model is based on a small number of constants, whose values have to be determined by running benchmarks. They can differ significantly between different installations – and even for different applications on one installation! This introduces a certain inaccuracy in the model.

**Collective Operations.** For sequential programs, there are by now more abstract programming models – first came structured programming, then functional and object-oriented programming. For parallel programs, abstractions are still being worked out. One small but rewarding step is to go from point-to-point communication, which causes unstructured communication code like the `goto` does control code in sequential programs [9], to patterns of communications and distributed computations. One frequently occurring case is the reduction, in which an associative binary operator is applied in a distributed fashion to a collection of values to obtain a result value (e.g., the sum or product of a sequence of numbers)<sup>4</sup>. A number of these operations are already provided by

---

<sup>4</sup> With a sequential loop, the execution time of a reduction is linear in the number of operand values. With a naive parallel tree computation, the time complexity is logarithmic, for a linear number of processors. This is not cost-optimal. However, with a commonly used trick, called Brent’s Theorem, the granularity of the parallelism can be coarsened, i.e., the number of processors needed can be reduced to maintain cost optimality in a shared-memory cost model [4].

MPI – programmers just need some encouragement to use them! The benefit of an exclusive use of collective operations is that the more regular structure of the program enables a better cost prediction. (BSP has a similar benefit.) With architecture-specific implementations of collective operations, one can calibrate program performance for a specific parallel computer [10].

One problem with libraries of high-performance modules is their lack of performance compositionality: the sequential composition of two calls of highly tuned implementations does, in general, not deliver the best performance. Better performance can be obtained when one provides a specific implementation for the composition of the two calls.

In general, it is hopeless to predict what compositions of calls users might require. But, at a comparatively low level of abstraction, e.g., at the level of collective operations, it is quite feasible to build a table of frequently occurring compositions and their costs incurred on a variety of parallel architectures. Gorlatch [11] and Kuchen [12] deal with this issue in their contributions to this volume.

**Skeleton Libraries.** As one raises the level of abstraction in libraries, it becomes common-place that a library module represents a pattern of computation and requires pieces of code to be passed as parameters, thus instantiating the pattern to an algorithm. Such a pattern of computation is also called a *template* or *skeleton*. One of the simplest examples is the reduction, which is one of the collective operations of MPI and which is also offered in HPF (see Sect. 6.2).

One major challenge is to find skeletons which are general enough for frequent use, the other is to optimize them for the context of their use.

Contributions to this volume by Bischof et al. [13] and by Kuchen [12] propose skeleton libraries for high-performance computing.

**More Abstract Libraries.** The library approach can be carried to arbitrarily high levels of abstraction.

For example, there are several very successful packages for parallel computations in linear algebra. Two libraries based on MPI are ScaLAPACK [14] and PLAPACK [15]. The latter uses almost exclusively collective operations. Both libraries offer some context sensitivity via accompanying structure parameters which provide information, e.g., about the dimensionality or shape (like triangularity or symmetry) of a matrix which is being passed via another parameter.

Another, increasingly frequent theme in high-performance parallelism is the paradigm of divide-and-conquer. Several skeleton libraries contain a module for parallel divide-and-conquer, and there exists a detailed taxonomy of parallel implementations of this computational pattern [16, 17].

**Class Libraries.** One popular need for a class library in high-performance computing is created by the language Java, which does not provide a dedicated data type for the multi-dimensional array – the predominant data structure used in high-performance computing. Java provides only one-dimensional arrays and allocates them on the heap. Array elements can be arrays themselves, which

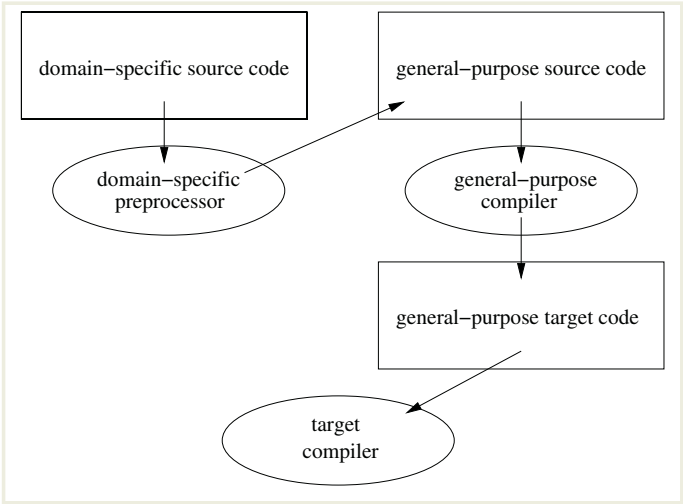
yields a multi-dimensional array, but these are not allocated contiguously and need not be of equal length. Thus, the resulting structure is not necessarily rectangular.

One reason why the multi-dimensional array is used so heavily in scientific computing is that it can be mapped contiguously to memory and that array elements can be referenced very efficiently by precomputing the constant part of the index expression at compile time [18]. There are several ways to make contiguous, rectangular, multi-dimensional arrays available in Java [19]. The easiest and most portable is via a class library, in which a multi-dimensional array is laid out one-dimensionally in row- or column-major order.

## 6 Preprocessors

### 6.1 Principle and Limitations

A preprocessor translates domain-specific language features into the general-purpose host language and, in the process, can perform a context-sensitive analysis, follow up with context-sensitive optimizations and generate more appropriate error messages (Fig. 2)<sup>5</sup>. For example, the structure parameters mentioned in the previous section could be generated more reliably by a preprocessor than by the programmer.



**Fig. 2.** Preprocessor

While this approach allows for a more flexible optimization (since there is no hard-wired case analysis), one remaining limitation is that no domain-specific

<sup>5</sup> Compile-time error messages can still be cryptic if they are triggered by the general-purpose code which the preprocessor generates.

optimizations can be performed below the level of the general-purpose host language. Only the general-purpose compiler can optimize at that level.

## 6.2 Examples from High-Performance Parallelism

**Minimal Language Extensions.** To put as little burden on the application programmer as possible, one can try to add a minimal extension to the sequential language. There are at least three commonly used languages which have been equipped this way. The internals of the compiler of the sequential language is not touched, but a parallel run-time system must be added. These language extensions are very easy to use. In particular, it is easy to parallelize existing programs which have been written without parallelism in mind<sup>6</sup>.

Cilk [20] is an extension of C for thread parallelism. It offers a good half dozen directives for spawning and synchronizing parallel threads and data access control. Cilk is based on a very smart run-time system for thread control and Cilk programs have won competitions and been awarded prizes (see the project's Web page).

Glasgow parallel Haskell (GpH) [21] is an extension of the functional language Haskell with just one function for spawning threads. Again, a special run-time system manages the threads.

JavaParty [22] is a preprocessor for Java to support the efficient computation with remote objects on clusters of processors. It adds one class attribute to Java: `remote`, which indicates that instances of the class can be moved to another processor. While Cilk and GpH are doing no preprocessing, JavaParty does. The preprocessor generates efficient code for object migration, finding a suitable location (processor), efficient serialization (coding the object for a fast transfer) and efficient garbage collection (exploiting the assumption of an un-failing network). An object may be migrated or cloned automatically, in order to increase the locality of accesses to it.

**HPF.** The most popular language extension in high-performance parallelism, supported by a preprocessor, is High-Performance Fortran (HPF) [23]. HPF is a modern version of sequential Fortran, enhanced with compiler directives. This cannot be considered a minimal extension, and one needs to have some expertise in parallelism to program HPF effectively. There are four types of directives: one for defining virtual processor topologies, one for distributing array data across such topologies, one for aligning arrays with each other, and one for asserting the independence of iterations of a loop.

HPF is at a much higher level of abstraction than Fortran with MPI. A program that takes a few lines in HPF often takes pages in Fortran with MPI<sup>7</sup>: the communication and memory management which is taken care of by the HPF compiler must be programmed explicitly in MPI.

---

<sup>6</sup> These days, we might call them “dusty disk programs”.

<sup>7</sup> E.g., take the example of a finite difference computation by Foster [24].

Most HPF compilers are preprocessors for a compiler for Fortran 90, which add calls to a library of domain-specific routines which, in turn, call MPI routines to maintain portability. One example is the HPF compiler ADAPTOR with its communications library DALIB [25].

There was justified hope for very efficient HPF programs. The general-purpose source language, Fortran, is at a level of abstraction which is pleasingly familiar to the community, yet comparatively close to the machine. Sequential Fortran is supported by sophisticated compilers, which produce highly efficient code. And, with the domain-specific run-time system for parallelism in form of the added library routines, one could cater to the special needs in the domain – also with regard to performance.

However, things did not quite work out as had been hoped. One requirement of an HPF compiler is that it should be able to accept every legal Fortran program. Since the HPF directives can appear anywhere and refer to any part of the program, the compiler cannot be expected to react reasonably to all directives. In principle, it can disregard any directive. The directives for data distributions are quite inflexible and the compiler's ability to deal with them depends on its capabilities of analyzing the dependences in the program and transforming the program to expose the optimal degree of parallelism and generate efficient code for it. Both the dependence analysis and the code generation are still areas of much research.

Existing HPF compilers deal well with fairly simple directives for scenarios which occur quite commonly, like disjoint parallelism or blockwise and cyclic data distributions. However, they are quite sensitive to less common or less regular dependence patterns: even when they can handle them, they often do not succeed in generating efficient code. Work on this continues in the area of loop parallelization (see further on).

With some adjustments of the paradigm, e.g., a more explicit and realistic commitment to what a compiler is expected to do and a larger emphasis on loop parallelization, a data-parallel Fortran might still have a future.

**OpenMP.** Most of HPF is *data-parallel*: a statement, which looks sequential, is being executed in unison by different processors on different sets of data. There are applications which are *task-parallel*, i.e., which require that different tasks be assigned to different processors. The preprocessor OpenMP [26] offers this feature (more conveniently than HPF) and is seeing quite some use – also, and to a major part, for SPMD parallelism. It is simpler to implement than an HPF preprocessor, mainly because it is based on the shared-memory model (while HPF is for distributed memory) and lets the programmer specify parallelism directly rather than via data distributions. On the downside, if the shared memory has partitions (as it usually does), the run-time system has the responsibility of placing the data and keeping them consistent.

**Loop Parallelization.** One way to make an HPF compiler stronger would be to give it more capabilities of program analysis. Much work in this regard has been devoted to the automatic parallelization of loop nests.

A powerful geometric model for loop parallelization is the *polytope model* [27, 28], which lays out the steps of a loop nest, iterating over an array structure, in a multi-dimensional, polyhedral coordinate space, with one dimension per loop. The points in this space are connected by directed edges representing the dependences between the loop iterations. With techniques of linear algebra and linear programming, one can conduct an automatic, optimizing search for the best mapping of the loop steps to time and space (processors) with respect to some objective function like the number of parallel execution steps (the most popular choice), the number of communications, a balanced processor load or combinations of these or others.

The polytope model comes with restrictions: the array indices and the loop bounds must be affine, and the space-time mapping must be affine<sup>8</sup>. Recent extensions allow mild violations of this affinity requirement – essentially, the permit a constant number of breaks in the affinity<sup>9</sup>. This admits a larger set of loop nests and yields better solutions.

Methods based on the polytope model are elegant and work well. The granularity of parallelism can also be chosen conveniently via a partitioning of the iteration space, called *tiling* [29, 30]. The biggest challenge is to convert the solutions found in the model into efficient code. Significant headway has been made recently on how to avoid frequent run-time tests which guide control through the various parts of the iteration space [31, 32].

Methods based on the polytope model have been implemented in various prototypical preprocessors. One for C with MPI is LooPo [33]. These systems use a number of well known schedulers (which compute temporal distributions) [34, 35] and allocators (which compute spatial distributions) [36, 37] for the optimized search of a space-time mapping. Polytope methods still have to make their way into production compilers.

**Preprocessing Library Calls.** There have been efforts to preprocess calls to domain-specific libraries in order to optimize the use of the library modules. One straight-forward approach is to collect information about the library modules in a separate data file and use a preprocessor to analyze the call context in the program and exploit the information given by rewriting the calls to ones which give higher performance. Such a preprocessor, the Broadway compiler [38] was used to accelerate PLAPACK calls by up to 30% by calling more specialized library modules when the context allowed.

In this approach, the library itself is neither analyzed nor recompiled. The correctness of the information provided is not being checked by the preprocessor. On the positive side, the approach can also be used when one does not have access to the source code of the library – provided the information one has about it can be trusted.

<sup>8</sup> An *affine function*  $f$  is of the form  $f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$ , where  $\mathbf{A}$  is a constant matrix and  $\mathbf{b}$  a constant vector. Affinity ensures the “straightness” of dependence paths across the iteration space of the loop nest, which allows the use of linear algebra and linear programming in their analysis.

<sup>9</sup> Dependence paths must still be straight but can change direction a fixed number of times.

## 7 Active Libraries

### 7.1 Principle and Limitations

The modules of an active library [39] are coded in two layers: the domain-specific layer, whose language offers abstractions for the special implementation needs of the domain, and the domain-independent layer in the host language. Thus, pieces of host code can be combined with domain-specific combinators<sup>10</sup>.

An active module can have several translations, each of which is optimized for a particular call context, e.g., a specific set of call parameters. The preprocessor is responsible for the analysis of the call context and for the corresponding translation of the domain-specific module code. The general-purpose compiler translates the pieces of host code which require no domain-specific treatment.

Arranging the program code in different layers, which are translated at different times or by different agents, is the principle of *multi-stage programming* (see the contribution of Taha [42] to this volume).

There is the danger of code explosion if a module is called in many different contexts.

### 7.2 Examples from High-Performance Parallelism

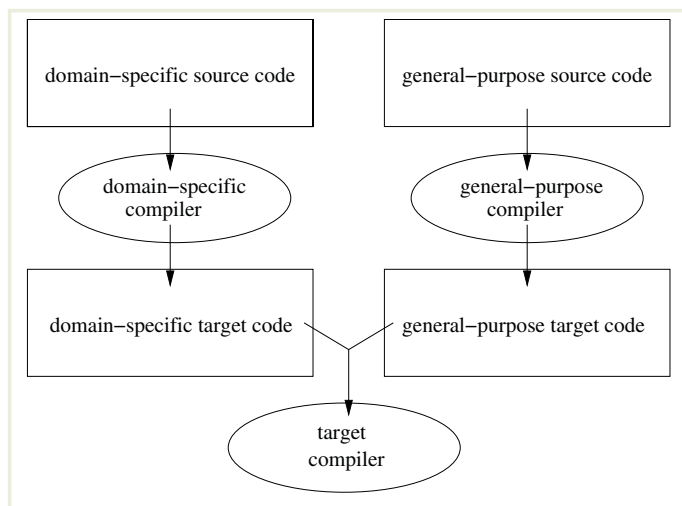
**Active Skeletons.** Herrmann has been experimenting with an active library approach for high-performance parallelism [41]. His domain-specific layer specifies the parallelism, synchronization and communication in the implementation of a skeleton. The domain-specific language is specially crafted to express structured parallelism. It is defined as an abstract data type in Haskell, and the preprocessor is written in Haskell. We have started with Haskell as the host language but have switched to C for performance reasons. C is also the target language.

**MetaScript.** The system Metascript, being proposed by Kennedy et al. [43], implements the idea of *telescoping languages*, which is similar to the idea of multi-staged, active libraries. Additional goals here are to achieve performance portability between different types of machines, to reduce compilation times and to facilitate fast, dynamic retuning at run time.

The modules of a library are annotated with properties and optimization rules. These are then analyzed and converted by a script generator to highly efficient code. The script is run to obtain the target code of the module for each type of machine and context. The target code still contains code for retuning the module at load time.

**The TaskGraph Library.** In their contribution to this volume, Beckmann et al. [44] describe an active library approach in which the generation of pieces of

<sup>10</sup> With *combinators* I simply mean operators that combine program parts. Examples in our domain are parallelism (`||`) and interleaving (`|||`) from CSP [40], and disjoint parallelism (`DPar`) and communicating parallelism (`ParComm`) from our own recent work [41].



**Fig. 3.** Two compilers

parallel code from an abstract specification (a syntax tree), and the adaptation of the target code to the context in which it appears, go on at run time. The advantage is, of course, the wealth of information available at run time. With an image filtering example, the authors demonstrate that the overhead incurred by the run-time analysis and code generation can be recovered in just one pass of an algorithm that iterates typically over many passes.

## 8 Two Compilers

### 8.1 Principle and Limitations

In order to allow context-sensitive, domain-specific optimizations below the level of the host language, one needs two separate compilers which both translate to the same target language; the two pieces of target code are then linked together and translated further by the target language compiler (Fig. 3). Note that, what is composed in sequence in Fig. 2, is composed unordered here.

There needs to be some form of information exchange between the general-purpose and the domain-specific side. This very important and most challenging aspect is not depicted in the figure because it could take many forms. One option is a (domain-specific) preprocessor which divides the source program into domain-specific and general-purpose code and provides the linkage between both sides.

The main challenge in this approach is to maintain a clean separation of the responsibilities of the two compilers:

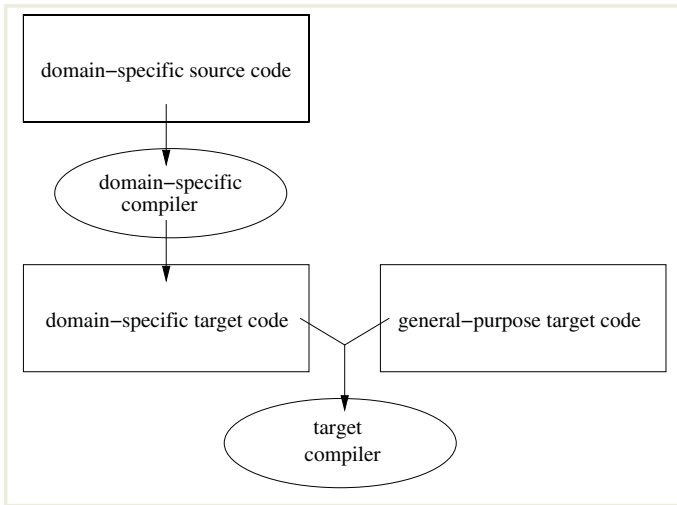
- The duplication of analysis or code generation effort by the two compilers should be minimized. One would not want to reimplement significant parts of the general-purpose compiler in the domain-specific compiler.



- No special demands should be made on the general-purpose compiler; only the target code it generates should be taken. As a consequence, it should be possible to upgrade the general-purpose compiler to a newer version without any changes of the domain-specific compiler.

## 8.2 Examples from High-Performance Parallelism

At present, I know of no case in which this approach is being pursued in high-performance parallelism. The problem is that high-performance programmers still (have to) cling to the host languages C and Fortran. These are suitable target languages in Fig. 3 but, in all approaches pursued so far, they are also the general-purpose host language. Thus, there is no general-purpose compiler and the domain-specific compiler becomes a preprocessor to the target compiler, and the picture degenerates to Fig. 4<sup>11</sup>. We have tried to use Haskell as general-purpose source language [41] but found that, with the C code generated by a common Haskell compiler, speedups are hard to obtain.



**Fig. 4.** Two compilers, degenerated

This situation will only improve if production compilers of more abstract languages generate target code whose performance is acceptable to the high-performance computing community. However, in some cases, a more abstract host language may aid prototyping, even if it does not deliver end performance.

The domains of linear algebra and of digital signal processing have been very successful in generating domain-specific optimizers, but with a focus on sequential programs. One notable project in linear algebra is ATLAS [45]; in

<sup>11</sup> Note the difference between Fig. 4 and Fig. 2: the level of abstraction to which the domain-specific side translates.

digital signal processing, there are FFTW [46] and SPIRAL [47]. FFTW comes close to the two-compiler idea and, since there is a parallel version of it (although this seems to have been added as an afterthought), I include it here.

**FFTW.** The FFTW project concentrates on the adaptive optimization of the fast Fourier transform. It follows a two-compiler approach, but with some departures from our interpretation of the scenario, as depicted in Fig. 3.

The idea of FFTW is to generate a discrete Fourier transform from a collection of automatically generated and highly optimized fragments (so-called *codelets*). The codelet generator corresponds to the general-purpose compiler in our picture. It is of a quite specialized form, so not really general-purpose. And it has been developed within the FFTW project, so it is not an external product. The source input it takes is simply an integer: the size of the transform to be calculated [48]. The codelet generator is written in Caml and produces platform-independent code in C. Actually, many users will just need to keep a bag of precompiled codelets and won't even have to install the codelet generator.

On the domain-specific side, codelet instances are selected and composed to a sequence by the so-called *plan generator*. The input to the plan generator consists of properties of the transform, like its size, direction and dimensionality. The search for the best sequence proceeds by dynamic programming, based on previously collected profiling information on the performance of the various plans. The sequence is then executed by the FFTW run-time system, the so-called *executor*.

In the case of FFTW, the only information passed explicitly between the two sides in our figure is the size parameter for the codelet generator, which is passed from the left to the right.

A parallel version of the executor has been implemented in Cilk.

## 9 Conclusions

So far, the main aims of domain-specific program generation seem to have been programming convenience and reliability. The perception of a need for domain-specific program optimization is just emerging.

Even in high-performance parallelism, an area with much work on domain-specific program optimization, most programmers favour programming at a low level. The easiest approach for today's programmer is to provide annotations of the simplest kind, as in Cilk, GpH and JavaParty, or of a more elaborate kind with HPF and OpenMP. Imposing more burden, but also offering more control over distributed parallelism and communication is MPI.

The contributions on high-performance parallelism in this volume are documenting an increasing interest in abstraction. A first step of abstracting from point-to-point communications in explicitly distributed programs is the use of collective operations (as provided by MPI). The next abstraction is to go to skeleton libraries as proposed by Bischof et al. [13] or Kuchen [12]. One step further would be to develop an active library.

The main issues that have to be addressed to gain acceptance of a higher level of abstraction in high-performance parallelism are:

- How to obtain or maintain full automation of the generative process? Here, the regularity of the code to be parallelized is counter-balanced by the deductive capabilities of the preprocessor. In loop parallelization one has to push the frontiers of the polytope model, in generative programming with active libraries one has to find subdomains which lend themselves to automation – or proceed interactively as, e.g., in the FAN skeleton framework [49].
- How to deal with the lack of performance compositionality? Unfortunately, it seems like this problem cannot be wished away. If they have the necessary knowledge, preprocessors can retune compositions of library calls (à la Gorlatch [11] and Kuchen [12]).
- How to identify skeletons of general applicability? The higher the level of abstraction, the more difficult this task can become if people have different ideas about how to abstract. Communication skeletons at the level of MPI and of collective operations are relatively easy. Also, for matrix computations, there is large agreement on what operations one would like. This changes for less commonly agreed-on structures like task farms, pipelines [50] and divide-and-conquer [16].

## Acknowledgements

Thanks to Peter Faber, Martin Griebel and Christoph Herrmann for discussions. Profound thanks to Don Batory, Albert Cohen and Paul Kelly for very useful exchanges on content and presentation. Thanks also to Paul Feautrier for many long discussions about domain-specific programming and program optimization. The contact with Paul Feautrier and Albert Cohen has been funded by a Procope exchange grant.

## References

1. Réveillère, L., Mérillon, F., Consel, C., Marlet, R., Muller, G.: A DSL approach to improve productivity and safety in device drivers development. In: Proc. Fifteenth IEEE Int. Conf. on Automated Software Engineering (ASE 2000), IEEE Computer Society Press (2000) 91–100
2. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* **35** (2000) 26–36
3. Hammond, K., Michaelson, G.: The design of Hume: A high-level language for the real-time embedded system domain (2004) In this volume.
4. Quinn, M.J.: *Parallel Computing*. McGraw-Hill (1994)
5. Robison, A.D.: Impact of economics on compiler optimization. In: Proc. ACM 2001 Java Grande/ISCOPE Conf., ACM Press (2001) 1–10
6. Pacheco, P.S.: *Parallel Programming with MPI*. Morgan Kaufmann (1997)
7. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press (1994)  
Project Web page: [http://www.csm.ornl.gov/pvm/pvm\\_home.html](http://www.csm.ornl.gov/pvm/pvm_home.html).

8. Skillicorn, D.B., Hill, J.M.D., McColl, W.F.: Questions and answers about BSP. *Scientific Programming* **6** (1997) 249–274  
Project Web page: <http://www.bsp-worldwide.org/>.
9. Gorlatch, S.: Message passing without send-receive. *Future Generation Computer Systems* **18** (2002) 797–805
10. Gorlatch, S.: Toward formally-based design of message passing programs. *IEEE Transactions on Software Engineering* **26** (2000) 276–288
11. Gorlatch, S.: Optimizing compositions of components in parallel and distributed programming (2004) In this volume.
12. Kuchen, H.: Optimizing sequences of skeleton calls (2004) In this volume.
13. Bischof, H., Gorlatch, S., Leshchinskiy, R.: Generic parallel programming using C++ templates and skeletons (2004) In this volume.
14. Blackford, L.S., Choi, J., Cleary, A., D’Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK: A linear algebra library for message-passing computers. In: *Proc. Eighth SIAM Conf. on Parallel Processing for Scientific Computing*, Society for Industrial and Applied Mathematics (1997) 15 (electronic) Project Web page: <http://www.netlib.org/scalapack/>.
15. van de Geijn, R.: Using PLAPACK: Parallel Linear Algebra Package. *Scientific and Engineering Computation Series*. MIT Press (1997) Project Web page: <http://www.cs.utexas.edu/users/plapack/>.
16. Herrmann, C.A.: The Skeleton-Based Parallelization of Divide-and-Conquer Recursions. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau (2001) Logos-Verlag.
17. Herrmann, C.A., Lengauer, C.: *HDC*: A higher-order language for divide-and-conquer. *Parallel Processing Letters* **10** (2000) 239–250
18. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers – Principles, Techniques, and Tools*. Addison-Wesley (1986)
19. Moreira, J.E., Midkiff, S.P., Gupta, M.: Supporting multidimensional arrays in Java. *Concurrency and Computation – Practice & Experience* **13** (2003) 317–340
20. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices* **33** (1998) 212–223 *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI’98)*. Project Web page: <http://supertech.lcs.mit.edu/cilk/>.
21. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + strategy = parallelism. *J. Functional Programming* **8** (1998) 23–60 Project Web page: <http://www.cee.hw.ac.uk/dsg/gph/>.
22. Philippsen, M., Zenger, M.: JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience* **9** (1997) 1225–1242 Project Web page: <http://www.ipd.uka.de/JavaParty/>.
23. Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele, Jr., G.L., Zosel, M.E.: *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press (1994)
24. Foster, I.: *Designing and Building Parallel Programs*. Addison-Wesley (1995)
25. Brandes, T., Zimmermann, F.: ADAPTOR—a transformation tool for HPF programs. In Decker, K.M., Rehmann, R.M., eds.: *Programming Environments for Massively Distributed Systems*. Birkhäuser (1994) 91–96
26. Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering* **5** (1998) 46–55 Project Web page: <http://www.openmp.org/>.

27. Lengauer, C.: Loop parallelization in the polytope model. In Best, E., ed.: CONCUR'93. LNCS 715, Springer-Verlag (1993) 398–416
28. Feautrier, P.: Automatic parallelization in the polytope model. In Perrin, G.R., Darte, A., eds.: The Data Parallel Programming Model. LNCS 1132. Springer-Verlag (1996) 79–103
29. Andonov, R., Balev, S., Rajopadhye, S., Yanev, N.: Optimal semi-oblique tiling. In: Proc.13th Ann. ACM Symp.on Parallel Algorithms and Architectures (SPAA 2001), ACM Press (2001)
30. Griebel, M., Faber, P., Lengauer, C.: Space-time mapping and tiling – a helpful combination. Concurrency and Computation: Practice and Experience **16** (2004) 221–246 Proc. 9th Workshop on Compilers for Parallel Computers (CPC 2001).
31. Quilleré, F., Rajopadhye, S., Wilde, D.: Generation of efficient nested loops from polyhedra. Int. J. Parallel Programming **28** (2000) 469–498
32. Bastoul, C.: Generating loops for scanning polyhedra. Technical Report 2002/23, PRISM, Versailles University (2002) Project Web page: <http://www.prism.uvsq.fr/~cedb/bastools/cloog.html>.
33. Griebel, M., Lengauer, C.: The loop parallelizer LooPo. In Gerndt, M., ed.: Proc. Sixth Workshop on Compilers for Parallel Computers (CPC'96). Konferenzen des Forschungszentrums Jülich 21, Forschungszentrum Jülich (1996) 311–320 Project Web page: <http://www.infosun.fmi.uni-passau.de/cl/loopo/>.
34. Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. Int. J. Parallel Programming **21** (1992) 313–348
35. Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. Int. J. Parallel Programming **21** (1992) 389–420
36. Feautrier, P.: Toward automatic distribution. Parallel Processing Letters **4** (1994) 233–244
37. Dion, M., Robert, Y.: Mapping affine loop nests: New results. In Hertzberger, B., Serazzi, G., eds.: High-Performance Computing & Networking (HPCN'95). LNCS 919. Springer-Verlag (1995) 184–189
38. Guyer, S.Z., Lin, C.: Optimizing the use of high-performance software libraries. In Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J., Pugh, W., Tseng, C.W., eds.: 13th Workshop on Languages and Compilers for Parallel Computing (LCPC 2000). LNCS 1717, Springer-Verlag (2001) 227–243
39. Czarnecki, K., Eisenecker, U., Glück, R., Vandevoorde, D., Veldhuizen, T.: Generative programming and active libraries (extended abstract). In Jazayeri, M., Loos, R.G.K., Musser, D.R., eds.: Generic Programming. LNCS 1766, Springer-Verlag (2000) 25–39
40. Hoare, C.A.R.: Communicating Sequential Processes. Series in Computer Science. Prentice-Hall Int. (1985)
41. Herrmann, C.A., Lengauer, C.: Using metaprogramming to parallelize functional specifications. Parallel Processing Letters **12** (2002) 193–210
42. Taha, W.: A gentle introduction to multi-stage programming (2004) In this volume.
43. Kennedy, K., Broom, B., Cooper, K., Dongarra, J., Fowler, R., Gannon, D., Johnson, L., Mellor-Crummey, J., Torczon, L.: Telescoping languages: A strategy for automatic generation of scientific problem solving systems from annotated libraries. J. Parallel and Distributed Computing **61** (2001) 1803–1826
44. Beckmann, O., Houghton, A., Mellor, M., Kelly, P.: Run-time code generation in C++ as a foundation for domain-specific optimisation (2004) In this volume.
45. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing **27** (2001) 3–35 Project Web page: <http://math-atlas.sourceforge.net/>.

46. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP'98). Volume 3. (1998) 1381–1384  
Project Web page: <http://www.fftw.org/>.
47. Püschel, M., Singer, B., Xiong, J., Moura, J.F.F., Johnson, J., Padua, D., Veloso, M., Johnson, R.W.: SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. J. High Performance in Computing and Applications (2003) To appear. Project Web page: <http://www.ece.cmu.edu/~spiral/>.
48. Frigo, M.: A fast Fourier transform compiler. ACM SIGPLAN Notices **34** (1999) 169–180 Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'99).
49. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: The FAN skeleton framework. Parallel Algorithms and Applications **16** (2001) 87–121
50. Kuchen, H., Cole, M.: The integration of task and data parallel skeletons. Parallel Processing Letters **12** (2002) 141–155

# A Personal Outlook on Generator Research

## (A Position Paper)

Yannis Smaragdakis

College of Computing, Georgia Institute of Technology  
Atlanta, GA 30332, USA  
yannis@cc.gatech.edu  
<http://www.cc.gatech.edu/~yannis/>

**Abstract.** If we want domain-specific program generation to form the basis of a strong, long-lived research community, we need to recognize what its potential impact might be and why the promise has not been fulfilled so far. In this chapter, I review my past work on generators and I present a collection of personal opinions on the symptoms convincing me that there is room for improvement in the generators research community. Then I analyze the causes of these symptoms, some of which are inherent, while some others can be overcome. A major cause of difficulty is the inherent domain-specificity of generators that often makes research work be less valuable to other generator writers who are unfamiliar with the domain. I propose directions on what should be considered promising research for the community, what I believe are useful principles for generator design, and what community building measures we can take.

## 1 Introduction

This chapter is a personal account of my past work and current thoughts on research in software generators and the generators research community.

As an opinion piece, this chapter contains several unsubstantiated claims and (hopefully) many opinions the reader will find provocative. It also makes liberal use of the first person singular. At the same time, whenever I use the first person plural, I try to not have it mean the “royal ‘we’ ” but instead to speak on behalf of the community of generators researchers.

There are two ways to view the material of this chapter. The first is as a threat-analysis for the area of domain-specific program generation. Indeed, a lot of the discussion is explicitly critical. For instance, although I believe that domain-specific program generation has tremendous potential, I also feel that the domain-specificity of the area can limit the potential for knowledge transfer and deep research. Another way to view this chapter, however, is as an opportunity-analysis: based on a critical view of the area, I try to explicitly identify the directions along which both research and community-building in software generators can have the maximum impact.

I will begin with a description of my background in generators research. This is useful to the reader mainly as a point of reference for my angle and outlook.

## 2 My Work in Generators

A large part of my past and present research is related to software generators. I have worked on two different transformation systems (Intentional Programming and JTS), on the DiSTiL generator, on the Generation Scoping facility, on C++ Template libraries and components, and on the GOTECH framework for generation of EJB code.

### 2.1 Transformation Systems

Transformation systems are infrastructure for generators. They are usually tools for doing meta-programming: writing programs that manipulate other programs. Transformation systems typically include a language extension facility (e.g., a macro language and ways to add concrete syntax), and a transformation engine: a way to specify manipulations of syntax entities and semantic information.

I have worked on two different transformation systems: Intentional Programming (IP) and JTS. IP [1, 11, 12] was a long term project at Microsoft Research that intended to provide a complete and mature language extensibility environment. IP's goal was to accommodate an ecology of transformations that would be provided by different sources and would not need to be designed to cooperate. (The term "ecology" refers to a biological metaphor that has transformations or language features be the analogue of genes and programming languages be the analogue of organisms, through which transformations propagate and evolution takes place.) As part of my work on IP, I implemented code template operators (a quote/unquote facility for manipulating program fragments as data) and a pattern language for transformations. Two interesting results of my IP work were the *generation scoping* facility and the *DiSTiL* generator, both of which I describe later in more detail.

JTS [3] is an extensible Java parser and syntax analyzer. It enables syntactic extensions to the language and arbitrary syntax tree transformations. JTS provides a standard programmatic interface for manipulating syntax trees, but also offers a full pattern language and a macro facility. JTS has served mostly as an experimentation platform for language extension ideas (e.g., additions of templates and module-like constructs in Java [17]) and domain-specific languages (e.g., the P3 generator [4]).

### 2.2 DiSTiL

The DiSTiL domain specific language [12] is an extension to C that allows the user to compose data structure components to form very efficient combinations of data structures. The language has a declarative syntax for specifying data structure operations: the user can define traversals over data through a predicate that the data need to satisfy. The DiSTiL generator can then perform optimizations based on the static parts of the predicate. Optimizations include the choice of an appropriate data structure, if multiple are available over the same data.



The following (slightly simplified) DiSTiL source code fragment shows the main elements of the language, including the data structure definition (`typeq1` and `cont1` definitions), cursor predicate (`curs1` definition) and traversal keywords (`foreach`, `ref`).

```
struct phonebook_record {...};    // C definition

typeq (phonebook_record, Hash(Tree(Malloc(Transient)))) typeq1;
Container (typeq1, (Hash (phone), Tree (name))) cont1;
Cursor (cont1, name > "Sm" && name < "Sn", ascending(name))
    curs1;                        // DiSTiL definitions

...
foreach(curs1)
    ... ref(curs1, name) ...
// DiSTiL operations mixed with C code
```

This example code shows a data structure organizing the same data in two ways: using a hash table (the `Hash` component in the above code) on the “phone” field and using a red-black tree (`Tree`) on the “name” field of the data records. The data are stored transiently in memory (`Transient`) and allocated dynamically on the heap (`Malloc`). The cursor shown in the example code is defined using a predicate on the “name” field. Therefore, DiSTiL will generate efficient code for all uses of this cursor by using the red-black tree structure. Should the cursor predicate or the data structure specification change in the future, DiSTiL will generate efficient code for the new requirements without needing to change the data structure traversal code.

### 2.3 Generation Scoping

Consider the generation of code using code template operators `quote` (`'`) and `unquote` (`$`). The generator code may contain, for instance, the expression:

```
'(if ((fp = fopen($filename, "r")) == NULL)...)'
```

The question becomes, what are the bindings of free variables in this expression? What is the meaning of `fp` or even `fopen`? This is the scoping problem for generated code and it has been studied extensively in the *hygienic macros* literature [5, 8] for the case of pattern-based generation. Generation scoping [16] is a mechanism that gives a similar solution for the case of programmatic (i.e., not pattern-based) generation. The mechanism adds a new type, `Env`, and a new keyword, `environment`, that takes an expression of type `Env` as an argument. `environment` works in conjunction with `quote` and `unquote` – all generated code fragments under an `environment(e)` scope have their variable declarations inserted in `e` and their identifiers bound to variables in `e`. For example, the following code fragment demonstrates the generation scoping syntax:

```

Env e = new Env(parent);
...
environment(e)
    return '{ FILE *fp; ... }'
...
environment(e)
    return '{ if ((fp = fopen($filename, "r")) == NULL)
              FatalError(FILE_OPEN_ERROR);
              ...
            }'

```

In the above example, a new environment, **e**, gets created. Then, a variable, **fp**, is added to it, just by virtue of quoting its declaration under **e**. Any subsequent code generated under environment **e** will have its identifiers bound to variables in **e**. For instance, the **fp** identifier in the last quoted code fragment will be bound to the **fp** variable generated earlier. This binding is ensured, even if there are multiple **fps** visible in the same lexical scope in the generated program, by consistently renaming the **fps** in the generated code to a unique name.

The advantage of generation scoping is that it ensures that scoping is what the generator programmer intends. The scoping of an identifier in a generated fragment is not determined merely by its location in the final generated program. This allows the arbitrary mixing and matching of generated code fragments without worrying about name conflicts. The sophisticated part of the generation scoping implementation is that it needs to recognize what parts of the generated code correspond to binding instances (e.g., the declaration **FILE \*fp**) and produce code that adds them in the right scoping environment maintained during generation. Environments can be organized hierarchically – in the above example, environment **e** is a child of environment **parent**. An identifier is looked up in the current environment, then this environment’s parent, etc. Hierarchies of environments allow generation scoping to mimic a variety of scoping arrangements (e.g., lexical scoping but also ad hoc namespaces) in the generated program.

Generation scoping simplified significantly the implementation of DiSTiL. DiSTiL is a component-based generator – the generated code is produced by putting together a large numbers of smaller code fragments. Thus, the same code is often generated in the same lexical scope but with different intended identifier bindings. This was made significantly easier by generation scoping, as each generation-time component only needed to maintain a single environment, regardless of how the code of all components ended up being weaved together.

## 2.4 C++ Templates Work

Advanced work with C++ templates is often closely related to generators. C++ templates offer a Turing-complete compile-time sub-language that can be used to perform complex meta-programming [6]. In combination with the C++ syntactic extensibility features, like operator overloading, C++ templates offer an extensible language environment where many domain-specific constructs can be

added. In fact, several useful implementations of domain-specific languages [19], especially for scientific computing, have been implemented exclusively as C++ template libraries. These domain-specific languages/libraries perform complex compile-time optimizations, akin to those performed by a high-performance Fortran compiler.

Much of my work involves C++ templates although often these templates are not used for language extensibility. Specifically, I have proposed the concept of a *mixin layer* [13, 14, 17]. Mixin layers are large-scale components, implemented using a combination of inheritance and parameterization. A mixin layer contains multiple classes, all of which have a yet-unknown superclass. What makes mixin layers convenient is that all their component classes are simultaneously instantiated as soon as a mixin layer is composed with another layer. Mixin layers can have a C++ form as simple as the following:

```
template <class S> class T : public S {
    class I1: public S::I1 {...};
    class I2: public S::I2 {...};
    class I3: public S::I3 {...};
};
```

That is, a mixin layer in C++ is a class template, *T*, that inherits from its template parameter, *S*, while it contains nested classes that inherit from the corresponding nested classes of *S*. In this way, a mixin layer can inherit entire classes from other layers, while by composing layers (e.g., *T<A>*) the programmer can form inheritance hierarchies for a whole set of inter-related classes (like *T<A>::I1*, *T<A>::I2*, etc.).

My C++ templates work also includes FC++ [9, 10]: a library for functional programming in C++. FC++ offers much of the convenience of programming in Haskell without needing to extend the C++ language. Although the novelty and value of FC++ is mostly in its type system, the latest FC++ versions make use of C++ template meta-programming to also extend C++ with a sub-language for expressing lambdas and various monad-related syntactic conveniences (e.g., comprehensions).

## 2.5 GOTECH

The GOTECH system is a modular generator that transforms plain Java classes into Enterprise Java Beans – i.e., classes conforming to a complex specification (J2EE) for supporting distribution in a server-side setting. The purpose of the transformation is to make these classes accessible from remote machines, i.e., to turn local communication into distributed. In GOTECH, the programmer marks existing classes with unobtrusive annotations (inside Java comments). The annotations contain simple settings, such as:

```
/**
 *@ejb:bean name = "SimpleClass"
 *           type = "stateless"
```

```
*      jndi-name = "ejb/test/simple"
*      semantics = "by-copy"
*/
```

From such information, the generator creates several pieces of Java code and meta-data conforming to the specifications for Enterprise Java Beans. These include remote and local interfaces, a deployment descriptor (meta-data describing how the code is to be deployed), etc. Furthermore, all clients of the annotated class are modified to now make remote calls to the new form of the class. The modifications are done elegantly by producing code in the AspectJ language [7] that takes care of performing the necessary redirection. (AspectJ is a system that allows the separate specification of dispersed parts of an application's code and the subsequent composition of these parts with the main code body.) As a result, GOTECH is a modular, template-based generator that is easy to change, even for end-users.

## 2.6 Current Work

Some of my yet unpublished work is also relevant to generators research and language extensibility. In particular, the MAJ system is a meta-programming extension to Java for creating AspectJ programs. Specifically, MAJ adds to Java code template operators (a quote and unquote constructs) for structured (i.e., statically syntax-checked) generation of AspectJ code. The application value of MAJ is that it allows the easy creation of generators by just producing AspectJ programs that will transform existing applications. In general, I believe that using aspect-oriented technologies, like AspectJ, as a back-end for generators is a very promising approach.

Another one of my current projects is LC++. LC++ extends C++ with a full logic sub-language, closely resembling Prolog. The extension is implemented using template meta-programming, i.e., as a regular C++ library without needing any modifications to existing compilers.

## 3 What Are the Difficulties of Generator Research?

After describing my past and current work in generators, I take off my researcher hat and put on my GPCE'03 PC co-Chair hat. The GPCE conference (Generative Programming and Component Engineering) is trying to build a community of people interested in work in program generation<sup>1</sup>. Clearly, I advertise GPCE in this chapter, but at the same time I am hoping to outline the reasons why I

---

<sup>1</sup> The name also includes "Component Engineering" which is a closely related area. The main element behind both generators and component engineering is the domain-specificity of the approaches. Some domains are simple enough that after the domain analysis is performed there is no need for a full-blown generator or language. Instead, an appropriate collection of components and a straightforward composition mechanism are a powerful enough implementation technique.

think GPCE is important and why the generators community needs something like what I imagine GPCE becoming. All of my observations concern not just GPCE but the overall generators community. At the same time, all opinions are, of course, only mine and not necessarily shared among GPCE organizers. When I speak of “generators conferences”, the ones I have in mind are ICSR (the International Conference on Software Reuse), GPCE, as well as the older events DSL (the Domain-Specific Languages conference), SAIG (the workshop on Semantics, Applications and Implementation of Program Generation), and GCSE (the Generative and Component-based Software Engineering conference). Of course, much of the work on generators also appears in broader conferences (e.g., in the Object-Oriented Programming or Automated Software Engineering communities) and my observations also apply to the generators-related parts of these venues.

Is there something wrong with the current state of research in generators or the current state of the generators scientific community? One can certainly argue that the community is alive and well, good research is being produced, and one cannot improve research quality with strategic decisions anyway. Nevertheless, I will argue that there are some symptoms that suggest we can do a lot better. These symptoms are to some extent shared by our neighboring and surrounding research communities – those of object-oriented and functional programming, as well as the broader Programming Languages and Software Engineering communities. I do believe, however, that some of the symptoms outlined below are unique and the ones that are shared are even more pronounced in the generators community. By necessity, my comments on community building are specific to current circumstances, but I hope that my comments on the inherent difficulties of generator research are general.

### 3.1 Symptoms

*Relying on Other Communities.* The generators community is derivative, to a larger extent than it should be. This means that we often expect technical solutions from the outside. The solution of fundamental problems that have direct impact to the generators community is often not even considered *our* responsibility. Perhaps this is an unfair characterization, but I often get the impression that we delegate important conceptual problems to the programming languages or systems communities. A lot of the interactions between members of the generators community and researchers in, say, programming languages (but outside generators) take the form of “what cool things did you guys invent lately that we can use in generators?”.

Although I acknowledge that my symptom description is vague, I did want to state this separately from the next symptom, which may be a cause as well as an expression of this one.

*Low Prestige.* The generators community is lacking in research prestige. Specific indications include the lack of a prestigious, high-selectivity publication outlet, and the corresponding shortage of people who have built a career entirely on

doing generators work. Most of us prefer to publish our best results elsewhere. Of course, this is a chicken-and-egg problem: if the publication outlets are not prestigious, people will not submit their best papers. But if people do not submit their best papers, the publication outlets will remain non-prestigious. I don't know if GPCE will overcome this obstacle, but I think it has a chance to do so. GPCE integrates both people who are interested in generators applications (the Software Engineering side) and people who work on basic mechanisms for generators (the Programming Languages side). GPCE is a research conference: the results that it accepts have to be new contributions to knowledge and not straightforward applications of existing knowledge. Nevertheless, research can be both scientific research (i.e., research based on analysis) and engineering research (i.e., research based on synthesis). Both kinds of work are valuable to GPCE. The hope is that by bringing together the Software Engineering and the Programming Languages part of the community, the result will be a community with both strength in numbers but also a lively, intellectually stimulating exchange of ideas.

*Poor Definition.* Another symptom suggesting that the generators community could improve is the vagueness of the limits of the community. Most research communities are dynamic, but I get the impression that we are a little more dynamic than the average. The generators conferences get a lot of non-repeat customers. Often, papers are considered relevant under the reasoning of “XYZ could be thought of as a generator”. Where do we draw the line? Every community has links to its neighboring communities, but at the same time a community is defined by the specific problems they are primarily interested in or the approach they take to solutions.

*Limited Impact.* A final, and possibly the most important, symptom of the problems of our community has to do with the impact we have had in practice. There are hundreds of nice domain-specific languages out there. There are several program generation tools. A well-known software engineering researcher recently told me (upon finding out I work on generators) “You guys begin to have impact! I have seen some very nice domain-specific languages for XYZ.” I was embarrassed to admit that I could not in good conscience claim any credit. Can we really claim such an impact? Or were all these useful tools developed in complete isolation from research in software generators? If we do claim impact, is it for ideas, for tools, or for methodologies? In the end, when a new generator is designed, domain experts are indispensable. Does the same hold for research results?

One can argue that this symptom is shared with the programming languages research community. Nevertheless, I believe the problem is worse for us. The designers of new general purpose programming languages (e.g., Java, C#, Python, etc.) may not have known the latest related research for every aspect of their design. Nevertheless, they have at least read some of the research results in language design. In contrast, many people develop useful domain-specific languages without ever having read a single research paper on generators.

### 3.2 Causes?

If we agree that the above observations are indeed symptoms of a problem, then what is the cause of that problem? Put differently, what are the general obstacles to having a fruitful and impactful research community in domain-specific program generation? I believe there are two main causes of many of the difficulties encountered by the generators community.

1. Domain-specificity is inherent to generators: most of the value of a generator is in capturing the domain abstractions. But research is all about transmission of knowledge. If the value is domain-specific, what is there to transmit to others?
2. What is generators work anyway? There is no established common body of knowledge for the area of generators. Consequently, it is not clear what are the big research problems and what should be the next research goals.

In the next two sections, I try to discuss in more detail these two causes. By doing this, I also identify what I consider promising approaches to domain-specific program generation research.

## 4 Domain-Specificity

### 4.1 Lessons That Transcend Domains

In generators conferences, one finds several papers that tell a similarly-structured tale: “We made this wonderful generator for domain XYZ. We used these tools.” Although this paper structure can certainly be very valuable, it often degenerates into a “here’s what I did last summer” paper. A domain-specific implementation may be valuable to other domain experts, but the question is, what is the value to other generators researchers and developers who are not domain experts? Are the authors only providing an example of the success of generators but without offering any real research benefit to others? If so, isn’t this really not a research community but a birds-of-a-feather gathering?

Indeed, I believe we need to be very vigilant in judging technical contributions according to the value they offer to other researchers. In doing this, we could establish some guidelines about what we expect to see in a good domain-specific paper. Do we want an explicit “lessons learned” section? Do we want authors to outline what part of their expertise is domain-independent? Do we want an analysis of the difficulties of the domain, in a form that will be useful to future generators’ implementors for the same domain? I believe it is worth selecting a few good domain-specific papers and using them as examples of what we would like future authors to address.

### 4.2 Domain-Independent Research: Infrastructure

In my view, a very promising direction of generators research is the design and development of infrastructure: language abstractions and type system support,

transformation systems, notations for transformations, etc. A lot of generators research, both on the programming languages and the software engineering side, is concerned with generator/meta-programming infrastructure. Infrastructure is the domain-independent part of generators research. As such, it can be claimed to be conceptually general and important to the entire generators community, regardless of domain expertise. I believe that the generators community has the opportunity (and the obligation!) to develop infrastructure that will be essential for developing future generators. In this way, no generator author will be able to afford to be completely unaware of generators research.

Of course, the potential impact of infrastructure work has some boundaries. These are worth discussing because they will help focus our research. The margin of impact for infrastructure is small exactly because domain-specificity is so inherent in generators work – domain knowledge is the quintessence of a generator. I usually think of the range with potential for generator infrastructure as a narrow zone between the vast spaces of the *irrelevant* and the *trivial*. Infrastructure is irrelevant when the domain is important and its abstractions mature. For domain specific languages like Yacc, Perl, SQL, etc., it does not matter what infrastructure one uses for their generators. The value of the domain is so high, that even if one invests twice as much effort in building a generator, the “wasted” effort will be hardly noticed. Similarly, infrastructure is sometimes trivial. A lot of benefit has been obtained for some domains by mere use of text templates. Consider Frame Processors [2] – a trivial transformational infrastructure with significant practical applications. Frame Processors are like low-level lexical macros. A more mature meta-programming technology is certainly possible, but will it matter, when Frame Processors are sufficient for getting most of the benefit in their domain of application?

Despite the inherent limitations of research in generator infrastructure, I believe the potential is high. Although the range between the irrelevant and the trivial is narrow, it is not narrower than the research foci of many other communities. After all, scientific research is the deep study of narrow areas. If the required depth is reached, I am hopeful that practical applications will abound. For example, a convenient, readily available, and well-designed meta-programming infrastructure for a mainstream language is likely to be used by all generator developers using that language.

### 4.3 Promising Infrastructure Directions

To further examine the promising directions for having real impact on generator development, it is worth asking why generators fail. I would guess (without any statistical basis) that for every 100 generators created, about one will see any popularity. The reasons for failure, I claim, are usually the small benefit (the generator is just a minor convenience), extra dependency (programmers avoid the generator because it introduces an extra dependency), and bad fit of the generator (the code produced does not fit the development needs well). Of these three, “small benefit” is a constant that no amount of research can affect – it is inherent in the domain or the understanding of the domain concepts by



the generator writer. The other two reasons, however, are variables that good generator infrastructure can change. In other words, good infrastructure can result in more successful generators.

Given that our goal is to help generators impose fewer dependencies and fit better with the rest of a program, an interesting question is whether a generator should be regarded as a tool or as a language. To clarify the question, let's characterize the two views of a generator a little more precisely. Viewing a generator as a language means treating it as a closed system, where little or no inspection of the output is expected. Regarding a generator as a tool means to support a quick-and-dirty implementation and shift some responsibility to the user: sometimes the user will need to understand the generated code, ensure good fit with the rest of the application, and even maintain generated code.

The two viewpoints have different advantages and applicability ranges. For example, when the generator user is not a programmer, the only viable option is the generator-as-a-language viewpoint. The generator-as-a-language approach is high-risk, however: it requires buy-in by generator users because it adds the generator as a required link in the dependency chain. At the same time, it implies commitment to the specific capabilities supported by the generator. The interconnectivity and debugging issues are also not trivial. In summary, the generator-as-a-language approach can only be valuable in the case of well-developed generators for mature domains. Unfortunately, this case is almost always in the irrelevant range for generator infrastructure. Research on generator infrastructure will very rarely have any impact on generators that are developed as languages. If such a generator is successful, its preconditions for success are such that they make the choice of infrastructure be irrelevant.

Therefore, I believe the greatest current promise for generator research with impact is on infrastructure for generators that follow the generator-as-a-tool viewpoint. Of course, even this approach has its problems: infrastructure for such generators may be trivial – as, for instance, in the case of the “wizards” in Microsoft tools that generate code skeletons using simple text templates. Nonetheless, the “trivial” case is rare in practice. Most of the time a good generator-as-a-tool needs some sophistication – at the very least to the level of syntactic and simple semantic analysis.

How can good infrastructure help generators succeed? Recall that we want to *reduce dependencies on generator tools* and *increase the fit of generated code to existing code*. Based on these two requirements, I believe that a few good principles for a generator-as-a-tool are the following:

- Unobtrusive annotations: the domain-specific language constructs should be in a separate file or appear as comments in a regular source file. The source file should be independently compilable by an unaware compiler that will just ignore the domain-specific constructs. Although this is overly restrictive for many domains, when it is attainable it is an excellent property to strive for.

- Separate generated code: generated code should be cleanly separated using language-level encapsulation (e.g., classes or modules). A generator should be a substitute for something the programmer feels they could have written by hand and does not pollute the rest of the application. The slight performance loss due to a compositional encapsulation mechanism should not be a concern. A generator-as-a-tool is foremostly a matter of high-level expressiveness and convenience for the programmer, not a way to apply very low-level optimizations, like inlining.
- Nice generated code: generated code should be well formatted, and natural (idiomatic) for the specific target language. This ensures maintainability.
- Openness and configurability: The generator itself should be written using standard tools and should even be changeable by its users! Code templates and pattern-based transformation languages are essential.

For instance, recall the DiSTiL generator that I mentioned in Section 2. DiSTiL is an extension to the C language. The DiSTiL keywords are obtrusive, however, and the DiSTiL generated code is weaved through the C code of the application for efficiency. I reproduce below the DiSTiL source code fragment shown earlier:

```
struct phonebook_record {...};    // C definition

typeq (phonebook_record, Hash(Tree(Malloc(Transient)))) typeq1;
Container (typeq1, (Hash (phone), Tree (name))) cont1;
Cursor (cont1, name > "Sm" && name < "Sn", ascending(name))
    curs1;                          // DiSTiL definitions

...
foreach(curs1)
    ... ref(curs1, name) ...
// DiSTiL operations mixed with C code
```

If I were to reimplement DiSTiL now, I would introduce all its definitions inside comments. All the generated code would use standard encapsulation features of the target language (e.g., classes in Java or C++) and instead of special traversal operations, like `foreach`, I would use the existing C++ STL idioms for collections. Essentially, DiSTiL would just be a generator for code that the programmer could otherwise write by hand. The dependency would be minimal – the programmer is always in control and can choose to discontinue use of the generator at any time. The source code could look like the following:

```
/* @Typeq Hash[phone](Tree[name](Malloc(Transient))) ContType1;
 * @Container ContType1 cont1;
 * @Cursor Curs1(cont1, name > "Sm" && name < "Sn",
 *               ascending(name));
 */
```

```

struct phonebook_record {...};
for (ContType1::Curs1 curs = cont1.begin();
     curs != cont1.end();
     curs++)
    ... curs->name ...
// C++ STL collection idiom

```

Clearly, good infrastructure can be invaluable in such a generator implementation. There need to be mature tools for parsing both the original source language code and the structured comments. Creating language extensions inside comments can also be facilitated by a special tool. Debugging of generated code is a thorny practical issue that can be alleviated only with good infrastructure. Making the generator itself be an open tool that its users can configure is essential. This can only be done if the generator is based on a standard meta-programming infrastructure, instead of being a one-time result. All of the above tasks are research and not straightforward development: the right design of all these language tools is very much an open problem.

In short, generators research *can* have impact through infrastructure. Even if the infrastructure is not instantly widely adopted, by designing it correctly we can hope that the generators the *do* adopt our infrastructure end up being the successful ones.

## 5 What Is Generators Work Anyway?

The second cause of many of the difficulties of the generators research community is the lack of consensus on what is generators research. The community does not have a well-defined background – there is no set of papers or textbooks that we all agree everybody should read before they do research in this area. Of course, the limits of what is considered topical for our research community are inherently vague: some of our research can fit the general programming languages community, while some more fits fine in traditional software engineering outlets. Nevertheless, I do not think this is the real problem. Multiple research communities have significant overlap. In fact, the generators community does have a very distinct research identity. Generators researchers work on very specific problems (e.g., language extensibility, meta-programming, domain-specific language design). Therefore, I believe that the lack of consensus on the core of generators research is a truly solvable problem, which just requires a lot of community vigilance!

It is perhaps interesting to briefly discuss *why* generators researchers do research in this area. What are the elements that attract us to this particular style of research and why do we feel the problems that we work on justify our research existence? I think that there are two main groups of people doing research in this area. The first consists of the people who are enthusiastic about generators as an intellectual challenge. The well-known argument for translation techniques applies perfectly to generators research: “If you are doing Computer Science, you probably think computations are cool. If so, then what can be cooler than

computing about computations?” There is a certain amount of excitement when one observes a program creating another program. Many generators researchers still feel the magic when they come across a self-reproducing program<sup>2</sup>.

The second group of people who do research in generators are those who see a lot of practical potential for simplifying programming tasks through automation. These researchers are primarily interested in the software engineering benefits of generators. Indeed, in my experience, the potential for automating software development is responsible for a steady supply of graduate students who aspire to make significant contributions to generators research. Most of them are quickly discouraged when they realize that automation of programming tasks is a well-studied, but extremely hard problem. Yet others join the ranks of generators researchers for good.

With this understanding of “who we are”, I believe we should try to establish a consistent identity as a research area. Again, this requires vigilance. But we can and should have some agreement across the community on issues like:

- What is the standard background in generators research? (This book largely tries to address this question.) What papers or books should most people be aware of?
- What are the elements of a good paper in the area? In the case of domain-specific work, how can we make sure it is valuable to non-domain-experts?
- What are some big open problems in generators? What are promising directions for high impact? (Batory’s “The Road to Utopia: A Future for Generative Programming”, in this volume, contains more discussion along these lines.)

Finally, we do need a prestigious, reliable publication outlet. Nothing defines a community better than a flagship publication channel. This will allow researchers to be permanently associated with the area, instead of seeking to publish strong results elsewhere.

## References

1. William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi, “Transformation in Intentional Programming”, in Prem Devanbu and Jeffrey Poulin (eds.), *proc. 5th International Conference on Software Reuse (ICSR ’98)*, 114-123, IEEE CS Press, 1998.
2. Paul Basset, *Framing Software Reuse: Lessons from the Real World*, Yourdon Press, Prentice Hall, 1997.
3. Don Batory, Bernie Lofaso, and Yannis Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages”, in Prem Devanbu and Jeffrey Poulin (eds.), *proc. 5th International Conference on Software Reuse (ICSR ’98)*, 143-155, IEEE CS Press, 1998.

---

<sup>2</sup> E.g., `((lambda (x) (list x (list 'quote x))) '(lambda (x) (list x (list 'quote x))))` in Lisp.

4. Don Batory, Gang Chen, Eric Robertson, and Tao Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Transactions on Software engineering*, 26(5), 441-452, May 2000.
5. William Clinger and Jonathan Rees, "Macros that work", *Eighteenth Annual ACM Symposium on Principles of Programming Languages (PoPL '91)*, 155-162, ACM Press, 1991.
6. Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Techniques, and Applications*, Addison-Wesley, 2000.
7. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, "An Overview of AspectJ", in Jørgen Lindskov Knudsen (ed.), proc. *15th European Conference on Object-Oriented Programming (ECOOP '01)*. In Lecture Notes in Computer Science (LNCS) 2072, Springer-Verlag, 2001.
8. Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba, "Hygienic macro expansion", in Richard P. Gabriel (ed.), proc. *ACM SIGPLAN '86 Conference on Lisp and Functional Programming*, 151-161, ACM Press, 1986.
9. Brian McNamara and Yannis Smaragdakis, "Functional programming in C++", in Philip Wadler (ed.), proc. *ACM SIGPLAN 5th International Conference on Functional Programming (ICFP '00)*, 118-129, ACM Press, 2000.
10. Brian McNamara and Yannis Smaragdakis, "Functional Programming with the FC++ Library", *Journal of Functional Programming (JFP)*, Cambridge University Press, to appear.
11. Charles Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", *NATO Science Committee Conference*, 1995.
12. Yannis Smaragdakis and Don Batory, "DiSTiL: a Transformation Library for Data Structures", in J. Christopher Ramming (ed.), *Conference on Domain-Specific Languages (DSL '97)*, 257-269, Usenix Association, 1997.
13. Yannis Smaragdakis and Don Batory, "Implementing Reusable Object-Oriented Components", in Prem Devanbu and Jeffrey Poulin (eds.), proc. *5th International Conference on Software Reuse (ICSR '98)*, 36-45, IEEE CS Press, 1998.
14. Yannis Smaragdakis and Don Batory, "Implementing Layered Designs with Mixin Layers", in Eric Jul (ed.), *12th European Conference on Object-Oriented Programming (ECOOP '98)*, 550-570. In Lecture Notes in Computer Science (LNCS) 1445, Springer-Verlag, 1998.
15. Yannis Smaragdakis and Don Batory, "Application Generators", in J.G. Webster (ed.), *Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons 2000.
16. Yannis Smaragdakis and Don Batory, "Scoping Constructs for Program Generators", in Krzysztof Czarnecki and Ulrich Eisenecker (eds.), *First Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, 65-78. In Lecture Notes in Computer Science (LNCS) 1799, Springer-Verlag, 1999.
17. Yannis Smaragdakis and Don Batory, "Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs", *ACM Trans. Softw. Eng. and Methodology (TOSEM)*, 11(2), 215-255, April 2002.
18. Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis and Marc Fleury, "Aspectizing Server-Side Distribution", in proc. *18th IEEE Automated Software Engineering Conference (ASE'03)*, 130-141, IEEE CS Press, 2003.
19. Todd Veldhuizen, "Scientific Computing in Object-Oriented Languages web page", <http://www.oonumerics.org/>

# Generic Parallel Programming Using C++ Templates and Skeletons

Holger Bischof<sup>1</sup>, Sergei Gorlatch<sup>1</sup>, and Roman Leshchinskiy<sup>2</sup>

<sup>1</sup> University of Münster, Germany  
`{hbischof,gorlatch}@uni-muenster.de`  
<sup>2</sup> Technical University of Berlin, Germany  
`rl@cs.tu-berlin.de`

**Abstract.** We study how the concept of generic programming using C++ templates, realized in the Standard Template Library (STL), can be efficiently exploited in the specific domain of parallel programming. We present our approach, implemented in the DatTeL data-parallel library, which allows simple programming for various parallel architectures while staying within the paradigm of classical C++ template programming. The novelty of the DatTeL is the use of higher-order parallel constructs, *skeletons*, in the STL-context and the easy extensibility of the library with new, domain-specific skeletons. We describe the principles of our approach based on skeletons, and explain our design decisions and their implementation in the library. The presentation is illustrated with a case study – the parallelization of a generic algorithm for carry-lookahead addition. We compare the DatTeL to related work and report both absolute performance and speedups achieved for the case study on parallel machines with shared and distributed memory.

## 1 Introduction

The domain of parallel programming is known to be error-prone and strongly performance-driven. A promising approach to cope with both problems is based on skeletons. The term *skeleton* originates from the observation that many parallel applications share a common set of computation and interaction patterns such as pipelines and data-parallel computations. The use of skeletons (as opposed to programming each application “from scratch”) has many advantages, such as offering higher-level programming interfaces, opportunities for formal analysis and transformation, and the potential for generic implementations that are both portable and efficient.

Domain-specific features must be taken into account when using general-purpose programming languages like C, C++, Java, etc. in the area of parallel programming. Modern C++ programming makes extensive use of *templates* to program generically, e. g. to design one program that can be simply adapted for various particular data types without additional re-engineering. This concept has been realized efficiently in the Standard Template Library (STL) [1], which is part of the standard C++ library. Owing to its convenient generic features

and efficient implementation, the STL has been employed extensively in many application fields by a broad user community.

Our objective is to enable an efficient execution of STL programs on parallel machines while allowing C++ programmers to stay in the world of STL programming. We concentrate on so-called function templates, which are parameterized patterns of computations, and use them to implement skeletons.

We introduce the DatTeL, a new data-parallel library that provides the user with parallelism options at two levels: (1) using annotations for sequential programs with function templates, and (2) calling parallel functions of the library. The presentation is illustrated using throughout the chapter one case study – carry-lookahead addition, for which we demonstrate the development from a sequential, STL-based solution to a parallel implementation.

The structure of the chapter, with its main contributions is as follows:

- We present the general concept of the DatTeL and the role distribution between application programmers, systems programmers and skeleton programmers in the program development process (Sect. 2).
- We introduce skeletons as patterns of parallelism, and formulate our case study – the carry-lookahead addition – in terms of skeletons (Sect. 3).
- We describe the concepts of the STL, show their correspondence to skeletons, and express the case study using the STL (Sect. 4).
- We present the architecture of the DatTeL library and show how it is used to parallelize the case study (Sect. 5).
- We discuss the main features of the DatTeL implementation (Sect. 6).
- We report measurements for our resulting DatTeL-program, which show competitive performance as compared to low-level sequential code, and indicate the potential for good parallel speedup in realistic applications on both shared-memory and distributed-memory platforms (Sect. 7).

Finally, we compare our approach with related work, draw conclusions and discuss possibilities for future research.

## 2 From STL and Skeletons Towards DatTeL

The C++ programming language becomes increasingly popular in the domain of high-performance computing. On the one hand, this is due to its execution model which allows the programmer to efficiently utilize the underlying hardware. Recently, however, C++’s support for implementing libraries which can optimize and generate code without changing the compiler has become even more important. In this section, we show how C++ is used for domain-specific program generation and outline our approach to parallel programming with C++.

### 2.1 Templates in C++

Originally, templates were introduced in C++ for writing generic programs and data structures. For instance, a generic vector can be implemented as

```
template<class T> class vector { ... };
```

The above definition defines a family of classes parametrized over the type variable `T`. In order to use `vector` it must be *instantiated*, i.e. applied to a concrete type:

```
vector<int> intvector;
vector<float *> ptrvector;
```

What sets C++ apart from other languages with similar constructs is the ability to define different implementations of generics depending on the types they are applied to. For instance, we might use a specialized implementation for vectors of pointers (a technique often used to avoid code bloat):

```
template<class T> class vector<T *> { ... };
```

With this definition, `ptrvector` declared before benefits from the more efficient implementation automatically and transparently to the user. Moreover, no run-time overhead is incurred as this kind of polymorphism is resolved at compile-time. The technique is not restricted to classes – algorithms can be specialized in a similar manner.

This example demonstrates two key features of C++ templates: unification on type parameters and statically evaluated computations on types. In fact, the C++ type system is basically a Turing-complete functional programming language interpreted by the compiler [2].

A wide range of template-based approaches to compile-time optimization and code generation make use of this property, most notably traits [3] for specifying and dispatching on properties of types and expression templates [4] for eliminating temporaries in vector and matrix expressions. These techniques have given rise to the concept of active libraries [5], which not only provide preimplemented components but also perform optimizations and generate code depending on how these components are used. Frequently, such libraries achieve very good performance, sometimes even surpassing hand-coded implementations – Blitz++ [6] and MTL [7] are two notable examples. More importantly, however, they provide an abstract, easy-to-use interface to complex algorithms without a large penalty in performance.

Not surprisingly, these advantages have made C++ an interesting choice for implementing libraries for parallel programming, a domain where both high performance and a sufficiently high level of abstraction are of crucial importance. For instance, templates are exploited in POOMA, a successful parallel framework for scientific computing [8]. In the following, we investigate how C++ templates can be used to implement a skeleton-based, data-parallel library which completely hides the peculiarities of parallel programming from the user while still allowing a high degree of control over the execution of the parallel program.

## 2.2 The Standard Template Library

The most popular use of templates to date has been the Standard Template Library, or STL, which uses templates to separate containers (such as vectors



and lists) from algorithms (such as finding, merging and sorting). The two are connected through the use of iterators, which are classes that know how to read or write particular containers, without exposing the actual type of those containers. As an example we give the STL implementation to compute the partial sums of a vector of integers.

```
vector<int> v;
partial_sum(v.begin(), v.end(), v.begin(), plus<int>());
```

The STL function `partial_sum` has four arguments, the first and second describing the range of the first input vector, the third pointing to the beginning of the result vector and the last pointing to the parameter function. The two methods, `begin()` and `end()` of the STL class `vector`, return instances of `vector::iterator`, marking the beginning and end of the vector. The STL template class `plus` implements the binary operation addition.

As part of the standard C++ library, the STL provides a uniform interface to a large set of data structures and algorithms. Moreover, it allows user-defined components to be seamlessly integrated into the STL framework provided they satisfy certain requirements or concepts. When implementing a data-parallel library which provides parallel data structures and operations on them, adhering to the STL interface decreases the learning time for the users and ultimately results in programs which are easier to understand and maintain. However, the STL interface has been designed for sequential programming and cannot be directly used in a parallel setting. Our goal is to study how the STL interface can be enhanced to support data parallelism.

### 2.3 Skeletons

The work on skeletons has been mostly associated with functional languages, where skeletons are modeled as higher-order functions. Among the various skeleton-related projects are those concerned with the definition of relatively simple, basic skeletons for parallel programming, and their implementation. For example, the two well-known list processing operators *map* and *reduce* are basic skeletons with inherent parallelism (*map* corresponds to an element-wise operation and *reduce* to a reduction).

The main advantage of using well-defined basic skeletons is the availability of a formal framework for program composition. This allows a rich set of program transformations (e. g. transforming a given program in a semantically-equivalent, more efficient one) to be applied. In addition, cost measures can be associated with basic skeletons and their compositions.

Existing programming systems based on skeletons include P3L [9], HDC [10], and others. Recently, two new libraries have been proposed. The skeleton library [11] provides both task- and data-parallel skeletons in form of C++ function templates rather than within a new programming language. The eSkel library [12] is an extension to MPI, providing some new collective operations and auxiliary functions.

## 2.4 Our approach: Towards the DatTeL

The main goal of our approach is to facilitate the implementation of domain-specific, parallel programs using the STL, without forcing the user to leave the familiar area of generic data structures and operations on them.

In this context, the *data-parallel programming model* [13] is a natural fit. The design of our DatTeL library relies heavily on this model, in which parallelism is expressed by applying the same computation to all elements of containers, such as vectors or lists. These computations are then executed simultaneously. Many well-known algorithms provided by the STL (e.g. `partial_sum`) are inherently data-parallel as long as certain restrictions, mainly on data dependencies, are met. The model's key feature, a single control flow shared by all nodes participating in a computation, enables a smooth transition from and interoperability with existing STL code while hiding low-level details such as synchronization and communication from the programmer.

To implement parallel programs using the DatTeL, the user calls STL functions which are overloaded for parallel containers. Annotating a vector with `par::` marks a container as parallel. Thus, the DatTeL implementation to compute the partial sums of a vector of integers is very similar to the STL implementation presented in Sect. 2.2.

```
par::vector<int> v;
partial_sum(v.begin(), v.end(), v.begin(), plus<int>());
```

Like the STL, the DatTeL is extensible at every level. A programmer can use the library to implement a parallel application, extend the library to provide better support for the task at hand, or port it to a different parallel architecture.

**Table 1.** The role distribution in implementing parallel programs

Person	Writes	Examples
Application Programmer	Application	carry-lookahead addition, convex hull
Skeleton Programmer	Skeletons	STL algorithms, divide-and-conquer
Systems Programmer	Parallel Layer	collectives, send-recv

Table 1 shows the different roles that can be assumed by programmers:

- The *application programmer* uses a high-level library (in our case, DatTeL) that extends the concepts and interfaces of the STL and consists of a number of ready-to-use building blocks for implementing parallel programs.
- The *skeleton programmer* is responsible for implementing generic skeletons and adding new ones. The need for new skeletons arises when implementing a specific application; such skeletons can often be reused in later projects. The DatTeL library supports the development of reusable algorithms by providing well-defined concepts and interfaces similar to those of the STL.

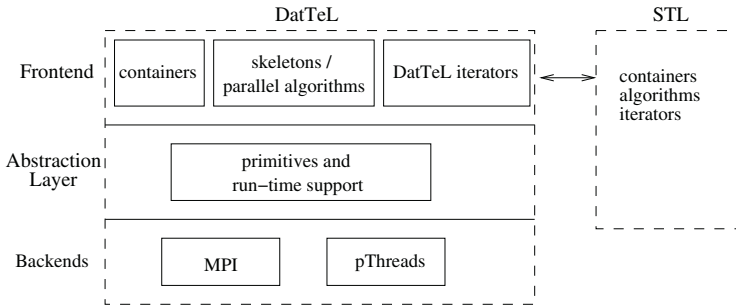
- The *systems programmer* ports the DatTeL to new architectures by implementing the backend, a small set of communication and synchronization primitives, in terms of a low-level communication library. The backend interface has been chosen carefully to ensure the library’s portability to a wide range of parallel architectures.

When faced with the task of implementing a new application, the DatTeL user can choose between two approaches:

1. Existing STL code can be reused and adapted to the parallel facilities provided by the DatTeL. The high degree of compatibility with the STL ensures that this is possible in many cases with little or no change to the code.
2. DatTeL-based programs can be developed from scratch. In this case, users can utilize their knowledge of STL programming styles and techniques and rapidly prototype the code by testing it with sequential algorithms and data structures provided by the STL and later replacing these with their parallel counterparts. We believe this to be a time- and cost-efficient approach to parallel programming.

## 2.5 The Architecture of the DatTeL

As shown in Fig. 1, the DatTeL has a modular, three-tier architecture geared to portability and ease of use.



**Fig. 1.** The DatTeL architecture

*Frontend.* Like the STL, the DatTeL library provides the programmer with a wide range of parallel data structures and algorithms. The DatTeL also adopts the STL’s notion of *concepts*, i.e. abstract, generic interfaces implemented by both predefined and user-defined components. Concepts such as iterators and sequences are generalized and modified to work in a parallel setting and new concepts such as data distributions are introduced.

*Abstraction layer.* The frontend components are built on top of a collection of parallel primitives and a small runtime library. This layer insulates the bulk of the DatTeL’s code from issues specific to the parallel systems and architectures which the library runs on and allows the DatTeL to remain generic and abstract.

*Backend.* Ultimately, the DatTeL relies on low-level libraries such as Pthreads or MPI that provide the communication and synchronization primitives needed to implement the DatTeL’s high-level functionality. The backend provides a uniform interface to these primitives. It currently supports both MPI and Pthreads.

### 3 Basic Skeletons and the Case Study

In this section, we present programming components available to the user of the DatTeL. These components are called *skeletons* and are introduced as language- and architecture-independent algorithmic patterns. In addition, we introduce our case study, carry-lookahead addition, and express it using skeletons.

A skeleton can be viewed formally as a higher-order function, customizable for a particular application by means of parameters provided by the application programmer. The first skeletons studied in the literature were traditional second-order functions known from functional programming: *zip*, *reduce*, *scan*, etc.

We confine ourselves to just two basic skeletons, which are needed for our case study. They are defined on non-empty lists, with concatenation as constructor. This provides a better potential for parallelization than usual lists with cons as constructor.

- *Zip*: component-wise application of a binary operator  $\oplus$  to a pair of lists of equal length

$$\text{zip}(\oplus)([x_1, \dots, x_n], [y_1, \dots, y_n]) = [(x_1 \oplus y_1), \dots, (x_n \oplus y_n)]$$

- *Scan-right*: computing prefix sums of a list by traversing the list from right to left and applying a binary operator  $\oplus$  (in this case one can also speak of suffix sums)

$$\text{scanr}(\oplus)([x_1, \dots, x_n]) = [(x_1 \oplus \dots \oplus (x_{n-2} \oplus (x_{n-1} \oplus x_n) \dots)), \dots, x_n]$$

These second-order functions are called “skeletons” because each of them describes a whole class of functions, obtainable by substituting application-specific operators for parameter  $\oplus$ .

Our two basic skeletons have an obvious data-parallel semantics; the asymptotic parallel complexity is constant for *zip* and logarithmic for the scan if  $\oplus$  is associative. However, if  $\oplus$  is non-associative, then the scan is computed sequentially with linear time complexity.

#### 3.1 Case Study: Carry-Lookahead Addition

Let us introduce our case study. We consider the addition of two  $N$ -bit integers,  $a$  and  $b$ , represented as bit vectors.

The classical addition algorithm scans over the bits of the two input values  $a$  and  $b$  from the least significant to the most significant bit. The  $i$ -th bit of the result is computed by adding the  $i$ -th bits of both input values,  $a_i$  and  $b_i$ , to the overflow of the less significant bit,  $o_{i-1}$ . An overflow bit  $o_i$  is set iff at least two of the three bits  $a_i, b_i, o_{i-1}$  are set. Obviously, this algorithm could be specified using the *scanr* skeleton. However, the binary operator of the scan is non-associative in this case, so it prescribes a strictly sequential execution.

To enable a parallel execution, we must rely on skeletons with associative base operators. Our next candidate is the carry-lookahead addition [14].

$a$	0 1 0 0 1 0	
$b$	0 1 1 1 0 0	
$spg = zip(get\_spg)(a, b)$	S G P P P S	$\left\{ \begin{array}{ll} \text{S} = \text{stop} & (a_i = b_i = 0) \\ \text{P} = \text{propagate} & (a_i \neq b_i) \\ \text{G} = \text{generate} & (a_i = b_i = 1) \end{array} \right.$
$spg' = snoc(spg, S)$	S G P P P S S	
$o = scanr(add\_op) spg'$	0 1 0 0 0 0 0	
$zip(add)(a, zip(add)(b, o))$	0 1 0 1 1 1 0	

**Fig. 2.** Example of Carry-lookahead addition

If  $o$  is the bit vector of overflows, the result of the addition is obtained by adding component-wise all three bit vectors  $a$ ,  $b$  and  $o$ , i.e. applying the *zip* skeleton with the bit addition (modulo 2) twice (see Fig. 2):

$$zip(add)(a, zip(add)(b, o)) \quad (1)$$

The bit vector of overflows is computed in two steps. First, we generate a temporary vector  $spg$  by applying component-wise an operation *get\_spg* to all bits of the input values  $a$  and  $b$ . If both bits are equal to 0, *get\_spg* yields S for stop, because no overflow will result even if the sum of the less significant bit generates an overflow. If one of the two input bits is set and the other not, *get\_spg* yields a P for propagate, because it results in an overflow iff the sum of the less significant bits generate an overflow. If both input bits are set, *get\_spg* yields G for generate, because it always results in an overflow regardless of the less significant bit (see Fig. 2). Vector  $spg$  is computed using the *zip* skeleton:

$$spg = zip(get\_spg)(a, b) \quad (2)$$

In the second step, we compute the vector of overflows  $o$  by scanning over the vector  $spg$ . If  $spg_{i-1}$  equals S (or G), then  $o_i$  will be 0 or 1, respectively. If it equals P, then we take the previous result  $o_{i-1}$  (see Fig. 2). To ensure that no overflow occurs at the least significant bit, we add an S on the right side of the vector  $spg$ . This operation is dual to the usual *cons* and therefore called *snoc*:

$$spg' = snoc(spg, S) \quad (3)$$

The vector of overflows  $o$  is computed by applying the *scanr* skeleton to *spg'*:

$$o = \text{scanr}(\text{add\_op}) \text{ spg}' \quad (4)$$

To obtain the overall result, we substitute (4) in (1):

$$\begin{aligned} CL\text{-add}(a, b) &= \text{zip}(\text{add})(a, \text{zip}(\text{add})(b, o)), \\ \text{where } o &= \text{scanr}(\text{add\_op})(\text{snoc}(\text{zip}(\text{get\_spg})(a, b), S)) \end{aligned} \quad (5)$$

The skeleton formulation (5), thus obtained by equational reasoning, has an important advantage over the classical algorithm: all its skeletons (*zip* and *scanr*) are parallelizable because *add\_op* is associative. In the next sections, we show how this language-independent formulation can be expressed in the C++ setting and then parallelized using the DatTeL.

## 4 From Skeletons to C++ Templates and the STL

In this section, we demonstrate that data-parallel skeletons are naturally expressible in the widely used C++ programming language using templates, and illustrate how our case study is programmed using the STL.

### 4.1 Skeletons vs. Templates

Skeletons are generic higher-order functions, i. e. they have parameter functions, whose type depends on the data type of the data structure. In C++, they can be implemented using the template mechanism. Complex functional parameters can be implemented using so-called functors, i. e. classes that implement the function call operator `operator()`. Skeletons usually do not depend on the exact representation of the container and can be applied to any data structure that provides a certain interface. For instance, sequential reduction can be defined for every container as long as individual elements of the container can be accessed in some way. Therefore, it is desirable to decouple such computations from the structure of the containers they operate upon.

The approach taken by the STL relies on *iterators*. An STL iterator points to an element of a container and provides operations for accessing the elements and for moving the iterator within the container. The exact operations supported by an iterator depend on the underlying data structure – only operations which can be implemented efficiently are provided. For instance, iterators over doubly linked lists can only be moved one element forward or backward, in contrast to vector iterators which can be moved an arbitrary number of elements. STL iterators are categorized correspondingly: doubly linked lists provide *bidirectional iterators* whereas vector iterators belong to the *random access iterator* category.

Usually, STL algorithms operate on sequences of elements described by pairs of iterators pointing to the beginning and the end of the sequence. For instance, the STL reduction takes a pair of iterators and a functor which is used for computing the result. Obviously, such algorithms correspond to our notion of

skeletons. In fact, the STL provides many skeletons known from functional programming, including reductions (`accumulate`), scans (`partial_sum`), map and zip (variants of `transform`). However, most of these algorithms work on *cons lists* and provide only sequential access to the elements. Parallel functionality is only supported by random access iterators as provided e.g. by the standard `vector` container. Thus, data-parallel skeletons can be implemented in an STL-like fashion as function templates taking pairs of random access iterators and functors.

## 4.2 Carry-Lookahead Addition: STL Implementation

Our skeleton-based representation (5) of the carry-lookahead addition uses only basic skeletons, for which there exist predefined functions in the STL. Thus, it can be directly expressed in the STL setting.

First, we implement the functional parameters `get_spg`, `add_op` and `add`:

```
const int STOP=0; const int GENERATE=1; const int PROPAGATE=2;
int get_spg(int a, int b) {
    if(a+b<1) return STOP;
    else if(a+b>1) return GENERATE;
    else return PROPAGATE;
}
int add_op(int a, int b) { return b==PROPAGATE?a:b; }
int binadd(int a, int b) { return (a+b)%2; }
```

Here, integer constants are used to represent `STOP`, `GENERATE` and `PROPAGATE`. All operator parameters and return values have the same type `int`, which enables a compact and efficient implementation.

```
vector<int> a(N); vector<int> b(N); vector<int> c(N+1);
// initialize vector a and b
transform(a.begin(), a.end(), b.begin(), c.begin()+1, get_spg);
c[0]=STOP;
partial_sum(c.begin(), c.end(), c.begin(), add_op);
transform(c.begin(), c.end()-1, a.begin(), c.begin(), binadd);
transform(c.begin(), c.end()-1, b.begin(), c.begin(), binadd);
```

**Fig. 3.** STL implementation of the carry-lookahead addition

The STL implementation is shown in Fig. 3. We use the STL functions `transform` for the *zip* skeleton and `partial_sum` for the *scanr* skeleton. The function `transform` has five arguments: the first two describe the range of the first input vector, the third and fourth point to the beginning of the second input vector and the result vector, and the last argument points to the parameter function. We store the bit vectors in `c`, which implements a growable array of type `int`. The first element of a vector corresponds to the least significant bit. The first element of vector `c` is initialized with `STOP` according to (3).

## 5 Programming with the DatTeL

In this section, we describe the design decisions made in the DatTeL and demonstrate how our case study can be implemented using the library.

### 5.1 Domain-Specific Programming Using the DatTeL

For the application programmer, porting existing STL code to the DatTeL is very simple: adding the namespace prefix `par::` to the container type names changes the code so that it works on the DatTeL's parallel containers instead of the STL's sequential ones. DatTeL algorithms (we use the STL notion of algorithm for higher-order functions on containers) on such parallel containers are for the most part syntactically and semantically equivalent to their sequential counterparts provided by the STL. However, when called on parallel containers, the computation of these algorithms will be parallel.

From the user's point of view, the DatTeL borrows the main generic features of the STL and provides the following advantages in the parallel setting:

- The parallelism is hidden in the DatTeL algorithms and remains invisible for the user. New skeletons are introduced and implemented by a skeleton programmer. All communication and synchronization is encapsulated in the backend which in particular provides collective operations known from MPI, e.g. broadcast, gather and scatter.
- Different parallel architectures are used in the same manner, owing to the common interface of the backend. This enables easy migration to different parallel systems. To port the DatTeL to a new parallel system, a systems programmer has to re-implement the backend for the new architecture.
- It is possible to combine different backends in one application, thereby taking advantage of hierarchical systems such as clusters of SMPs. For example, a vector can be distributed across the nodes of a cluster, while its subvectors are managed by multiple threads within an SMP node.

### 5.2 DatTeL as a Domain-Specific Version of the STL

A key feature of the DatTeL is an STL-like interface which facilitates the transition from sequential to parallel programming for C++ developers. However, the STL was originally designed as a purely sequential library. A number of important STL aspects cannot be transferred directly to the domain of parallel programming. For instance, a strictly left-to-right evaluation strategy is usually specified for collective operations, thus precluding their efficient parallel implementation. A major challenge is to identify requirements and concepts which the DatTeL cannot support without sacrificing efficiency and to modify or replace them suitably. Here, we follow Stroustrup's famous guideline [15] and "try to stay as close to the STL as possible but no closer".

Two design aspects of the STL are not adopted in the DatTeL. Firstly, instead of assuming a left-to-right evaluation order, the DatTeL leaves the order



unspecified. This allows for an efficient, tree-like parallel implementation of operations like reductions and scans. Secondly, the STL assumes a flat memory model, i.e. every element of a container can be accessed at any time. In parallel programs, this cannot be implemented efficiently without excessive synchronization. Therefore, the DatTeL does not support accessing arbitrary elements of parallel containers except in collective operations. These provide two kinds of iterators, *local* and *global* ones. Conceptually, local iterators can only point to an element which is owned by the current node. They can be dereferenced freely and meet the STL iterator requirements. In contrast, global iterators can point to an arbitrary container element without being STL iterators. Instead, we introduce *parallel iterator categories* which are similar to their sequential counterparts (cf. Sect. 4) but do not require a dereferencing operation. This approach leads to a leaner and more efficient implementation of the library and makes the parallel behaviour of applications more explicit and easier to predict.

### 5.3 Carry-Lookahead Addition: Parallelization

We are now ready to parallelize the STL implementation of the carry-lookahead addition described in Sect. 4.2. As the implementation presented in Fig. 3 uses only skeletons already provided by the STL, the user parallelizes the implementation by simply annotating `par::` to the containers (and, of course, adding initialization and finalization of the parallel library):

```
par::init();
par::vector<int> a(N), b(N), c(N+1);
// ...
par::finalize();
```

This program uses a standard backend, which in our current implementation of the DatTeL is the MPI library. To change the parallel library, e.g. to POSIX threads, the user simply adds a template parameter to the container:

```
typedef par::pthread PL;

PL::init(NTHREADS);
par::vector<int,PL> a(N), b(N), c(N+1);
// ...
PL::finalize();
```

Note that the method call `a.begin()` returns a DatTeL iterator pointing to the first element of the distributed container. The DatTeL overloads the STL algorithms – `transform` and `partial_sum` in our example – for DatTeL iterators. Thus, the function calls in our example automatically call the parallel implementation of the skeletons.

Note that if we have to compute the prefix sums sequentially on a parallel vector, e.g. if the base operation is not associative, then the parallel vector needs to be copied into a temporary sequential one (`std::vector`) and then back after the computation.

## 6 The Implementation of the DatTeL

In this section, we outline our approach to the implementation of the DatTeL library, concentrating on the design aspects discussed briefly in Sect. 2.

The most important skeletons, currently implemented in the DatTeL, are listed in Table 2. We group them in four rows: basic skeletons, instances of divide-and-conquer, composition of skeletons, and skeletons predefined in the STL. The first column shows the name of the skeleton typically used by the skeleton community, in the second column the name of the DatTeL function represents the corresponding skeleton, and the third column describes additional restrictions as compared to the DatTeL counterparts, e.g. the result of `partial_sum` is well-defined iff the binary operator is associative.

**Table 2.** Example skeletons provided by the DatTeL

class	skeleton	DatTeL function	remark
basic skeletons	<i>map</i> <i>zip</i>	variant of <code>transform</code> , <code>for_each</code> variant of <code>transform</code>	no side-effects in operation
	<i>reduce</i> <i>scan</i>	<code>accumulate</code> <code>partial_sum</code>	binary operator has to be associative
Instances of divide & conquer		<code>partition</code> <code>sort</code>	
composition of skeletons	<i>zip; reduce</i> <i>zip; zip</i>	<code>inner_product</code> <code>transform3</code>	
STL algorithms implementing skeletons		<code>copy</code> <code>find_if</code> <code>replace</code>	

Additionally, the DatTeL contains some generic divide-and-conquer skeletons, which are used to implement `partition` and `sort`.

### 6.1 Adding New Skeletons

While the DatTeL implements a number of primitive skeletons and includes several high-level ones, our goal is not to support a large but fixed number of predefined skeletons but to facilitate the development of new ones. This is achieved by providing basic building blocks for implementing complex computations, by making the backend interfaces sufficiently high-level and by including rigorous specifications of interfaces between the library's different layers.

There are two ways to implement new complex skeletons: (1) in terms of simpler ones which are already available, or (2) by invoking backend operations directly if the predefined algorithms do not provide the required functionality. Obviously, the first way is preferable since it allows the programmer to work at a more abstract level. We will now demonstrate briefly how to add a new skeleton to the DatTeL. Let us analyze the last step of the carry-lookahead addition in

which the input vectors and the overflow vector are summed componentwise. This was accomplished using the *zip* skeleton twice in (1). Alternatively, we can define a new skeleton, *zip3*, that takes three lists and a ternary operator:

$$\text{zip3}(f)([x_1, \dots, x_n], [y_1, \dots, y_n], [z_1, \dots, z_n]) = [f(x_1, y_1, z_1), \dots, f(x_n, y_n, z_n)]$$

Using a new operation *add3*, which returns the sum of three values, the two *zips* from (1) can be expressed as  $\text{zip3}(\text{add3})(a, b, o) = \text{zip}(\text{add})(a, \text{zip}(\text{add})(b, o))$ .

Let us discuss how the new skeleton *zip3* can be implemented as the template function `transform3` in the DatTeL. We start with a sequential implementation:

```
template< /* typenamees */ >
Out transform3(In1 first1, In1 last1, In2 first2, In3 first3,
               Out result, TerOp op) {
    for ( ; first1 != last1; ++first1, ++first2, ++first3, ++result)
        *result = op(*first1, *first2, *first3);
    return result;
}
```

In this example, `Out`, `In1`, `In2` and `In3` are type variables for iterator types and `TerOp` for a ternary operator. We omit most typenamees for the sake of brevity.

From the STL code, we proceed to the implementation in the DatTeL. The DatTeL's abstraction layer provides the function `makepar` that calls a given function with parameters on all processors, so we simply call `makepar` with the sequential version `std::transform3` and the parameters specified by the user.

```
template< /* typenamees */ >
Out transform3(In first, In last, In2 first2,
               In3 first3, Out res, TerOp op) {
    return makepar(std::transform3, first, last,
                  first2, first3, res, op);
}
```

We can use the new DatTeL function in the implementation of our case study, presented in Fig. 3. The latter two transforms can be substituted by

```
transform3(c.begin(), c.end()-1, a.begin(), b.begin(),
          c.begin(), binadd3);
```

Using `transform3` makes the implementation of our case study more obvious, because its semantics coincides better with the description of the carry-lookahead addition: “add all three vectors element-wise”. This more intuitive version is also more compact and has better performance (see Sect. 7).

## 6.2 Marshalling

In the distributed memory setting, efficient marshaling, i.e. the packing of objects by the sender and their unpacking by the receiver during communication, is of crucial importance. Implementing marshaling is a non-trivial task in any programming language. Unfortunately, C++'s lack of introspection facilities complicates a generic implementation of marshaling. In the DatTeL, we intend to

adopt an approach similar to that of the TPO++ library [16]. Here, a traits class is used to specify the marshaling properties of a class. When communicating a class instance, the user can choose between several strategies:

- use the bit pattern representing the object for communication,
- provide a template-based, compile-time data structure which describes the structure of the class and from which the library generates suitable marshaling functions,
- provide user-defined functions which pack the object into a buffer before communication and unpack it afterwards.

Since the decision about strategy can be made at compile time, this flexibility incurs no run-time overhead. Different container classes, such as the STL's **vector** and **list**, will use different strategies depending on the types of their elements. At the moment, only the first strategy is supported.

### 6.3 Vector Alignment

On distributed memory machines, algorithms like **transform3** which work on several vectors simultaneously must assign a node to each individual computation and make sure that the arguments to the computation are available locally. Consider, for instance, the case study from Sect. 4.2 which frequently performs an element-wise addition on two vectors, storing the results in a third one. A simple rule like “owner computes” is easy to implement but might be inefficient even in this simple case. If both arguments are stored on the same node, it is preferable to perform the addition on that node and then communicate the result, rather than communicating two arguments – this approach reduces the communication costs by a factor of two. However, if the result of the computation is a complex data structure, then “owner computes” is the better solution. In the final version of the library, users will be able to specify the communication costs for a type (again, using the traits technique). The library will then select an appropriate alignment strategy based on that specification. While not attempting to choose an optimal alignment of vectors – this would require a whole-program analysis, something a library cannot do, – we believe that this approach will significantly reduce communication costs in common cases.

### 6.4 Distribution of Data

The DatTeL's vector class is parametrized with a policy which specifies the distribution strategy for the vector, i.e. a type parameter is used to specify which nodes own and process the vector's elements. This efficient design technique is enjoying increasing popularity in the C++ community [17]. Using a statically resolved template parameter in this case allows the DatTeL algorithms to handle common cases more efficiently. To support complex distribution strategies, algorithms usually perform additional computations to determine the required communication pattern. This overhead is unnecessary for simple (e.g. regular)

distributions and should be avoided in such cases. This is achieved by statically selecting the appropriate version of the algorithm depending on the distribution policies of the vectors involved in the computation. Currently, the DatTeL only supports regular block distributions. We are working on implementing cyclic and irregular block distributions known from HPF. Eventually, we also hope to incorporate more complex distribution strategies, e.g. based on per-element workloads, which aid the user in implementing efficient load balancing.

## 6.5 Nested Parallelism

An efficient way of implementing nested parallelism is the flattening transformation [13, 18] which turns nested parallel computations into flat ones. Until now, only compiler-based approaches for flattening have been studied [19, 20]. As outlined in Sect. 2.1, C++ templates can be used to implement compile-time transformations on suitably formulated expressions. We plan to make use of this technique to provide support for nested data parallelism in the DatTeL.

## 7 Experimental Results

The critical questions about any new approach in the domain of parallel programming are: 1) whether the target performance can compete with hand-crafted solutions, and 2) how well does it scale on parallel machines.

1. Our first goal is to assess the overhead caused by the genericity of the STL and the DatTeL. We compare our STL and DatTeL implementations of the carry-lookahead addition with a hand-crafted, non-generic one:

```
for(int i=0; i<N; i++) c[i+1]=get_spg(a[i],b[i]); // forall
c[0]=STOP;
for(int i=0; i<N; i++)                                // parallel scan
    if(c[i+1]==PROPAGATE) c[i+1]=c[i];
for(int i=0; i<N; i++) c[i]=(a[i]+b[i]+c[i])%2; // forall
```

Note that if we are not restricted to the STL, the two zips in (5) can be implemented in one forall loop. Table 3 compares the runtimes of three sequential versions: 1) DatTeL on one processor, 2) STL, and 3) the non-generic version. We measured the STL and DatTeL versions both with `transform` and `transform3`.

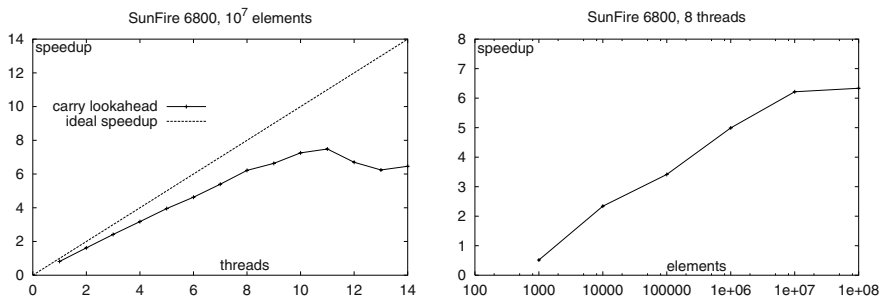
**Table 3.** Runtime in ms of the sequential carry-lookahead addition using a SunFire6800 with UltraSparc-III+ 900 MHz processors and a Cray T3E with Alpha 300 MHz.

	data size	DatTeL transform	DatTeL transform3	STL transform	STL transform3	non-generic version
SunFire 6800	$10^8$	23768	20632	23463	19097	16957
Cray T3E	$5 \cdot 10^6$	1124	897	1107	883	788

The abstraction penalty introduced by generic STL programming amounts to 12% of the non-generic version's runtime (non-generic vs. STL version using `transform3`). The overhead introduced by the DatTeL is less than 8% as compared to the STL version. We used the native C++ compiler 3.5.0.1 on the Cray and the gcc 3.2 on the SunFire 6800 machines both with optimization level `-O3`.

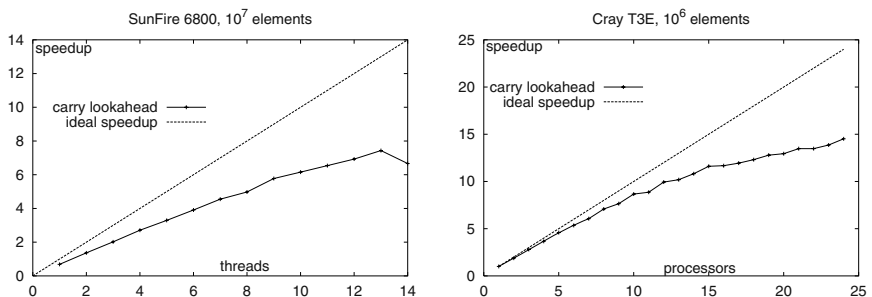
2. Our second goal is to assess parallel performance on different classes of parallel machines – with shared and distributed memory – and also to study the behaviour of DatTeL programs when the parallel machine is time-shared with other applications. We conducted our experiments with the parallelized carry-lookahead addition using the DatTeL library on the following machines:

- a dedicated shared-memory machine: SunFire 6800 with 12 processors of type UltraSparc-III+ 900 MHz and 24 GB of main memory, which was used exclusively for our measurements. The results for a fixed problem size are shown in Fig. 4 (left) and for a fixed number of threads in Fig. 4 (right)



**Fig. 4.** Speedups for carry-lookahead addition on a SunFire with 12 processors

- a non-dedicated shared-memory machine: SunFire 6800 with 16 processors of type UltraSparc-III 750 MHz and 32 GB of main memory, used in the time-shared mode: an average of four processors was used by other applications. The results for a fixed problem size are presented in Fig. 5 (left)



**Fig. 5.** Speedups for carry-lookahead addition; left: on a SunFire 6800 with 16 processors used as a server at a university computing-center; right: on a Cray T3E

- a dedicated distributed-memory machine: Cray T3E with 25 processors of type Alpha 300 MHz and 128 MB of main memory at each node, used exclusively for our measurements. The results for a fixed problem size are presented in Fig. 5 (right).

Our experiments demonstrate the good performance of the DatTeL for a large problem size. The achieved parallel speedup does not increase if the number of threads exceeds the number of available processors. Although the 16-processor machine in Fig. 5 (left) was in simultaneous use by other applications we still obtained a reduction of execution time when using up to 12 threads.

## 8 Conclusion and Related Work

Our work can be viewed as the adaptation of general-purpose C++ programming using the STL for the specific domain of parallel programming. The data-parallel template library DatTeL provides C++ programmers with various parallelism options while allowing them to stay in the world of STL programming. Most STL programs can be parallelized by simply annotating `par::` to the containers. The programmer can adapt a program to a new parallel system by adding a template parameter to the containers. This is facilitated by the three-tier architecture of the DatTeL and is reflected in the three different roles played by the groups of programmers: application, skeleton and systems programmer.

The DatTeL is, to the best of our knowledge, the first attempt to combine the STL with skeletons. In fact, the STL's generic algorithms themselves can be viewed as skeletons, albeit rather primitive ones. The DatTeL library has parallel semantics and provides a number of additional high-level skeletons, such as divide-and-conquer.

Another novel feature of our approach is that the DatTeL is easily extensible with new skeletons. This is achieved by introducing the abstraction layer, a well-defined interface between skeletons and low-level communication libraries, thus facilitating and encouraging the development of new skeletons. A new domain-specific skeleton can be added by composing already implemented skeletons or by using the abstraction layer.

We considered the carry-lookahead addition as our case study. It demonstrates the three-step process of domain-specific program development in the DatTeL: 1) find a skeleton representation of the problem, 2) directly express it in the STL, and 3) annotate the containers with `par::`.

Finally, we presented experimental results for programs using the DatTeL on parallel machines with both shared and distributed memory. Our time measurements demonstrate both competitive absolute performance of the DatTeL-based solution and its quite good parallel scalability.

For an overview of related approaches to parallel programming using C++, see [21]. There has been active research on implementations of skeletal approaches in C and C++ [11, 12, 22, 23]. Unlike these approaches, our DatTeL library aims to resemble the STL as much as possible. Two previous approaches extending the STL for parallel programming include: (1) the Parallel Standard

Template Library (PSTL) [24], which is part of the High Performance C++ Library, and (2) the Standard Template Adaptive Library (STAPL) [25]. Compared with PSTL and STAPL, the novel feature of the DatTeL is its use of an extensible set of skeletons. The DatTeL also differs in offering the potential of nested data parallelism, which was not covered here because of lack of space.

Our ongoing work on the DatTeL and future plans include:

- In addition to the available blockwise data distribution, we plan to implement also cyclic and block-cyclic distribution.
- Our current efforts include the implementation of matrix computations and simulations such as the Barnes-Hut algorithm.
- The use of the MPI backend is currently restricted to containers consisting of simple data types. Ongoing work adds marshalling and serialization of user-defined data types, such as classes, to the DatTeL.

It would be desirable to combine the data parallelism provided by the DatTeL with task parallelism. We plan to develop a task-parallel library that can be used together with the DatTeL, rather than to extend the DatTeL in the direction of task parallelism, because there are no close matches for it in the STL library.

An important question to be studied is performance portability: to describe how the target performance behaves depending on the machine used, we are developing a suitable cost model that is more precise than asymptotic estimates used in the paper. We are working on a cost calculus and performance prediction for DatTeL-based programs, based on the results achieved for skeletons and nested data parallelism in the context of the Nepal project [26].

## Acknowledgments

We are grateful to Chris Lengauer and two anonymous referees for many helpful comments and suggestions, and to Phil Bacon and Julia Kaiser-Mariani who assisted in improving the presentation.

## References

1. Stepanov, A., Lee, M.: The Standard Template Library. Technical Report HPL-95-11, Hewlett-Packard Laboratories (1995)
2. Veldhuizen, T.: Using C++ template metaprograms. C++ Report **7** (1995) 36–43 Reprinted in C++ Gems, ed. Stanley Lippman.
3. Myers, N.: Traits: a new and useful template technique. C++ Report (1995)
4. Veldhuizen, T.: Expression templates. C++ Report **7** (1995) 26–31
5. Veldhuizen, T.L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98), SIAM Press (1998)
6. Veldhuizen, T.L.: Arrays in Blitz++. In: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98). LNCS, Springer-Verlag (1998)



7. Siek, J.G., Lumsdaine, A.: The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In: ISCOPE. (1998) 59–70
8. Karmesin, S. et al.: Array design and expression evaluation in POOMA II. In Caromel, D., Oldehoeft, R., Tholburn, M., eds.: Computing in Object-Oriented Parallel Environments: Second International Symposium, ISCOPE 98. LNCS 1505, Springer-Verlag (1998)
9. Danelutto, M., Pasqualetti, F., Pelagatti, S.: Skeletons for data parallelism in P3L. In Lengauer, C., Griebel, M., Gorlatch, S., eds.: Euro-Par'97. Volume 1300 of LNCS., Springer (1997) 619–628
10. Herrmann, C.A., Lengauer, C.: HDC: A higher-order language for divide-and-conquer. *Parallel Processing Letters* **10** (2000) 239–250
11. Kuchen, H.: A skeleton library. In Monien, B., Feldmann, R., eds.: Euro-Par 2002. Volume 2400 of LNCS., Springer (2002) 620–629
12. Cole, M.: eSkel library home page. (<http://www.dcs.ed.ac.uk/home/mic/eSkel>)
13. Blelloch, G.E.: Programming parallel algorithms. *Communications of the ACM* **39** (1996) 85–97
14. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann Publ. (1992)
15. Koenig, A., Stroustrup, B.: As close as possible to C – but no closer. *The C++ Report* **1** (1989)
16. Grundmann, T., Ritt, M., Rosenstiel, W.: TPO++: An object-oriented message-passing library in C++. In: International Conference on Parallel Processing. (2000) 43–50
17. Alexandrescu, A.: Modern C++ Design. Addison-Wesley (2001)
18. Chakravarty, M.M.T., Keller, G.: More types for nested data parallel programming. In Wadler, P., ed.: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), ACM Press (2000) 94–105
19. Blelloch, G.E., Chatterjee, S., Hardwick, J.C., Sipelstein, J., Zagha, M.: Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing* **21** (1994) 4–14
20. Pfannenstiel, W. et al.: Aspects of the compilation of nested parallel imperative languages. In Werner, B., ed.: Third Working Conference on Programming Models for Massively Parallel Computers, IEEE Computer Society (1998) 102–109
21. Wilson, G., Lu, P., eds.: Parallel Programming using C++. MIT press (1996)
22. Dabrowski, F., Loulergue, F.: Functional bulk synchronous programming in C++. In: 21<sup>st</sup> IASTED International Multi-conference, AI 2003, Symposium on Parallel and Distributed Computing and Networks, ACTA Press (2003) 462–467
23. Danelutto, M., Ratti, D.: Skeletons in MPI. In Aki, S., Gonzales, T., eds.: Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems, ACTA Press (2002) 392–397
24. Johnson, E., Gannon, D.: Programming with the HPC++ Parallel Standard Template Library. In: Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, Minneapolis, SIAM (1997)
25. Rauchwerger, L., Arzu, F., Ouchi, K.: Standard templates adaptive parallel library. In O'Hallaron, D.R., ed.: 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers. LNCS 1511, Springer (1998) 402–410
26. Lechtchinsky, R., Chakravarty, M.M.T., Keller, G.: Costing nested array codes. *Parallel Processing Letters* **12** (2002) 249–266

# The Design of Hume: A High-Level Language for the Real-Time Embedded Systems Domain

Kevin Hammond<sup>1</sup> and Greg Michaelson<sup>2</sup>

<sup>1</sup> School of Computer Science,  
University of St Andrews, St Andrews, Scotland  
Tel: +44 1334 463241, Fax: +44 1334 463278  
`kh@dcs.st-and.ac.uk`

<sup>2</sup> Dept. of Mathematics and Computer Science,  
Heriot-Watt University, Edinburgh, Scotland  
Tel: +44 131 451 3422, Fax: +44 131 451 3327  
`greg@macs.hw.ac.uk`

**Abstract.** This chapter describes the design of Hume: a domain-specific language targeting real-time embedded systems. Hume provides a number of high level features including higher-order functions, polymorphic types, arbitrary but sized user-defined data structures, asynchronous processes, lightweight exception handling, automatic memory management and domain-specific meta-programming features, whilst seeking to guarantee strong space/time behaviour and maintaining overall determinacy.

## 1 The Real-Time Embedded Systems Domain

Over the last decade, embedded systems have become a fundamental part of everyday society in the form of systems such as automotive engine control units, mobile telephones, PDAs, GPS receivers, washing machines, DVD players, or digital set-top boxes. In fact, today more than 98% of *all* processors are used in embedded systems [32]. The majority of these processors are relatively slow and primitive designs with small memory capabilities. In 2002, 75% of embedded processors used 8-bit or 16-bit architectures and a total of a few hundreds of bytes is not uncommon in current micro-controller systems.

The restricted capabilities of embedded hardware impose strong requirements on both the space and time behaviour of the corresponding firmware. These limited capabilities are, in turn, a reflection of the the cost sensitivity of typical embedded systems designs: with high production volumes, small differences in unit hardware cost lead to large variations in profit. At the same time software production costs must be kept under control, and time-to-market must be minimised. This is best achieved by employing appropriate levels of programming abstraction.

### 1.1 Domain-Specific versus General-Purpose Language Design

Historically, much embedded systems software/firmware was written for specific hardware using native assembler. Rapid increases in software complexity and

the need for productivity improvement means that there has been a transition to higher-level *general-purpose* languages such as C/C++, Ada or Java. Despite this, 80% of all embedded systems are delivered late [11], and massive amounts are spent on bug fixes: according to Klocwork, for example, Nortel spends on average \$14,000 correcting each bug that is found once a system is deployed. Many of these faults are caused by poor programmer management of memory resources [29], exacerbated by programming at a relatively low level of abstraction. By adopting a *domain-specific* approach to language design rather than adapting an existing *general-purpose* language, it is possible to allow low-level system requirements to guide the design of the required high level language features rather than being required to use existing general-purpose designs. This is the approach we have taken in designing the Hume language, as described here.

Embedding a domain-specific language into a general purpose language such as Haskell [25] to give an *embedded domain-specific language* has a number of advantages: it is possible to build on existing high-quality implementations and to exploit the host language semantics, defining new features only where required. The primary disadvantage of the approach is that there may be a poor fit between the semantics of the host language and the embedded language. This is especially significant for the real-time domain: in order to ensure tight bounds in practice as well as in theory, all language constructs must have a direct and simple translation. Library-based approaches suffer from a similar problem. In order to avoid complex and unwanted language interactions, we have therefore designed Hume as a stand-alone language rather than embedding it into an existing design.

## 1.2 Language Properties

McDermid identifies a number of essential or desirable properties for a language that is aimed at real-time embedded systems [23].

- *determinacy* – the language should allow the construction of determinate systems, by which we mean that under identical environmental constraints, all executions of the system should be *observationally equivalent*;
- *bounded time/space* – the language must allow the construction of systems whose resource costs are statically bounded – so ensuring that *hard real-time* and *real-space* constraints can be met;
- *asynchronicity* – the language must allow the construction of systems that are capable of responding to inputs as they are received without imposing total ordering on environmental or internal interactions;
- *concurrency* – the language must allow the construction of systems as communicating units of independent computation;
- *correctness* – the language must allow a high degree of confidence that constructed systems meet their formal requirements [1].

Moreover, the language design must incorporate constructs to allow the construction of embedded software, including:

- *exception handling* to deal with runtime error conditions;
- *periodic scheduling* to ensure that real-time constraints are met;
- *interrupts and polling* to deal with connections to external devices.

Finally, we can identify a number of high-level features that assist program construction and reduce overall software costs:

- *automatic memory management* eliminates errors arising from poor manual memory management;
- *strong typing* eliminates a large number of programming errors;
- *higher-order functions* abstract over common patterns of computation;
- *polymorphism* abstracts internal details of data structures;
- *recursion* allows a number of algorithms, especially involving data structures, to be expressed in a natural, and thus robust fashion.

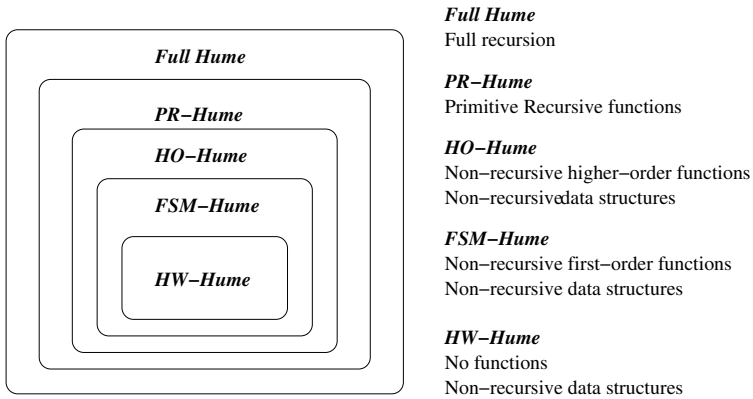
### 1.3 The Hume Design Model

The Hume language design attempts to maintain the essential properties and features required by the embedded systems domain outlined above (especially for transparent time and space costing) whilst incorporating as high a level of program abstraction as possible. We aim to target applications ranging from simple micro-controllers to complex real-time systems such as Smart Phones. This ambitious goal requires us to incorporate both low-level notions such as interrupt handling, and high-level ones of data structure abstraction etc. Of course such systems will be programmed in widely differing ways, but the language design should accommodate these varying requirements.

We have designed Hume as a three-layer language [15]: an outer (static) declaration/metaprogramming layer, an intermediate coordination layer describing a static layout of dynamic processes and the associated devices, and an inner layer describing each process as a (dynamic) mapping from patterns to expressions. The inner layer is stateless and purely functional. Rather than attempting to apply cost modelling and correctness proving technology to an existing language framework either directly or by altering a more general language (as with e.g. RTSj [7]), our approach is to design Hume in such a way that we are certain that formal models and proofs can be constructed. We envisage a series of overlapping Hume language levels as shown in Figure 1, where each level adds expressibility to the expression semantics, but either loses some desirable property or increases the technical difficulty of providing formal correctness/cost models.

## 2 Boxes and Coordination

Figure 2 shows the (simplified) syntax of Hume. *Boxes* are the key structuring construct. They are abstractions of finite state machines mapping tuples of inputs to tuples of outputs. The mapping is defined in a functional style, relating input patterns to output expressions. For example, we can define a box that acts like a binary and-gate as:

**Fig. 1.** Hume Design Space

```

box band in ( b1, b2 :: bit ) out ( b :: bit)
match (1,1) -> 1
|      (_,_) -> 0;

```

Although the body of a box is a single function, the *process* defined by a box will iterate indefinitely, repeatedly matching inputs and producing the corresponding outputs, in this case a single stream of bits representing a binary-and of the two input streams. Since a box is stateless, information that is preserved between box iterations must be passed explicitly between those iterations through some *wire* (Section 2.1). This roughly corresponds to tail recursion over a stream in a functional language [22], as recently exploited by E-FRP, for example [33]. In the Hume context, this design allows a box to be implemented as an uninteruptible thread, taking its inputs, computing some result values and producing its outputs [15]. Moreover, if a bound on dynamic memory usage can be predetermined, a box can execute with a fixed size stack and heap without requiring garbage collection [14].

## 2.1 Wiring

Boxes are connected using wiring declarations to form a static process network. A wire provides a mapping between an output link and an input link, each of which may be a named box input/output, a port, or a stream. Ports and streams connect to external devices, such as parallel ports, files etc. For example, we can wire two boxes `band` and `bor` into a static process network linked to the corresponding input and streams as follows:

```

box band in ( b1, b2 :: bit ) out ( b :: bit) ...
box bor in ( b1, b2 :: bit ) out ( b :: bit) ...

stream input1 from "idev1"; stream input2 from "idev2";
stream input3 from "idev3"; stream output to "odev";

```

```

wire input1 to band.b1; wire input2 to band.b2;
wire band.b to bor.b1; wire input3 to bor.b2;
wire bor.b to output;

initial band.b = 1;

```

<i>program</i> ::=	<i>decl</i> <sub>1</sub> ; ... ; <i>decl</i> <sub><i>n</i></sub>	<i>n</i> ≥ 1
<i>decl</i> ::=	<i>box</i>   <i>function</i>   <i>datatype</i>   <i>exception</i>   <i>wire</i>   <i>device</i>   <i>operation</i>	
<i>function</i> ::=	var <i>matches</i>	
<i>datatype</i> ::=	<b>data</b> id α <sub>1</sub> ... α <sub><i>m</i></sub> = <i>constr</i> <sub>1</sub>   ...   <i>constr</i> <sub><i>n</i></sub>	<i>n</i> ≥ 1
<i>constr</i> ::=	con τ <sub>1</sub> ... τ <sub><i>n</i></sub>	<i>n</i> ≥ 1
<i>exception</i> ::=	<b>exception</b> id [ :: τ ]	
<i>wire</i> ::=	<b>wire</b> <i>link</i> <sub>1</sub> <b>to</b> <i>link</i> <sub>2</sub> [ <b>initially</b> <i>cexpr</i> ]	
<i>link</i> ::=	<i>connection</i>   <i>deviceid</i>	
<i>connection</i> ::=	<i>boxid</i> . <i>varid</i>	
<i>box</i> ::=	<b>box</b> <i>id ins outs fair/unfair matches</i> [ <b>handle</b> <i>exnmatches</i> ]	
<i>ins/outs/ids</i> ::=	( <i>id</i> <sub>1</sub> , ... , <i>id</i> <sub><i>n</i></sub> )	
<i>matches</i> ::=	<i>match</i> <sub>1</sub>   ...   <i>match</i> <sub><i>n</i></sub>	<i>n</i> ≥ 1
<i>match</i> ::=	( <i>pat</i> <sub>1</sub> , ... , <i>pat</i> <sub><i>n</i></sub> ) → <i>expr</i>	
<i>expr</i> ::=	<i>int</i>   <i>float</i>   <i>char</i>   <i>bool</i>   <i>string</i>   var   *   con <i>expr</i> <sub>1</sub> ... <i>expr</i> <sub><i>n</i></sub>   ( <i>expr</i> <sub>1</sub> , ... , <i>expr</i> <sub><i>n</i></sub> )   <b>if</b> <i>cond</i> <b>then</b> <i>expr</i> <sub>1</sub> <b>else</b> <i>expr</i> <sub>2</sub>   <b>let</b> <i>valdecl</i> <sub>1</sub> ; ... ; <i>valdecl</i> <sub><i>n</i></sub> <b>in</b> <i>expr</i>   <i>expr</i> <b>within</b> ( <i>time</i>   <i>space</i> )	<i>n</i> ≥ 0 <i>n</i> ≥ 2
<i>valdecl</i> ::=	<i>id</i> = <i>expr</i>	
<i>pat</i> ::=	<i>int</i>   <i>float</i>   <i>char</i>   <i>bool</i>   <i>string</i>   var   _   *   _*   con var <sub>1</sub> ... var <sub><i>n</i></sub>   ( <i>pat</i> <sub>1</sub> , ... , <i>pat</i> <sub><i>n</i></sub> )	<i>n</i> ≥ 0 <i>n</i> ≥ 2
<i>device</i> ::=	( <b>stream</b>   <b>port</b>   <b>fifo</b>   <b>memory</b>   <b>interrupt</b> ) <i>devdesc</i>	
<i>devdesc</i> ::=	<i>id</i> [ ( <b>from</b>   <b>to</b> ) <i>string</i> [ <b>within</b> <i>time</i> <b>raising</b> <i>id</i> ] ]	
<i>operation</i> ::=	<b>operation</b> <i>id as string</i> :: τ	

Fig. 2. Hume Syntax (Simplified)

Note the use of an initialiser to specify that the initial value of the wire connected to `band.b` is 1.

## 2.2 Asynchronous Coordination Constructs

The two primary Hume constructs for asynchronous coordination are [15] to *ignore* certain inputs/outputs and to introduce *fair matching*. For example, a template for a fair merge operator can be defined as:

```
template merge ( mtype )
in ( xs :: mtype, ys :: mtype)
out ( xys :: mtype)
fair
  (x, *) -> x
| (*, y) -> y;
```

This matches two possible (polymorphic) input streams `xs` and `ys`, choosing fairly between them to produce the single merged output `xys`. Variables `x` and `y` match the corresponding input items in each of the two rules. The `*`-pattern indicates that the corresponding input position should be ignored, that is the pattern matches any input on the corresponding wire, without consuming it. Such a pattern must appear at the top level. Section 3.4 includes an example where an output is ignored.

## 3 Exceptions, Timing and Devices

### 3.1 Exceptions

Exceptions are raised in the expression layer and handled by the surrounding box. In order to ensure tight bounds on exception handling costs, exception handlers are not permitted within expressions. Consequently, we avoid the hard-to-cost chain of dynamic exception handlers that can arise when using non-real-time languages such as Java or Concurrent Haskell [26]. The following (trivial) example shows how an exception that is raised in a function is handled by the calling box. Since each Hume process instantiates one box, the exception handler is fixed at the start of each process and can therefore be called directly. A static analysis is used to ensure that all exceptions that could be raised are handled by each box. We use Hume's `as` construct to coerce the result of the function `f` to a string containing 10 characters.

```
exception Div0 :: string 10

f n = if n == 0 then raise Div0 "f" else (n / 0) as string 10;

box example in (c :: char) out (v :: string 29) handles Div0
match n -> f n
handle Div0 x -> "Divide by zero in: " ++ x;
```

### 3.2 Timing

Hume uses a hybrid static/dynamic approach to modelling time. Expressions and boxes are costed statically using a time analysis (as outlined in Section 4 for FSM-Hume programs). Since it is not possible to infer costs for arbitrarily complex full Hume programs, we provide a mechanism for the programmer to introduce time information where this has been derived from other sources, or to assert conditions on time usage that must be satisfied at runtime. Dynamic time requirements can be introduced at two levels: inside expressions, the `within` construct is used to limit the time that can be taken by an expression. Where an expression would exceed this time, a `timeout` exception is raised. A fixed time exception result is then used instead. In this way we can provide a hard guarantee on execution time. At the coordination level, timeouts can be specified on both input and output wires, and on devices. Such timeouts are also handled through the exception handler mechanism at the box level. This allows hard real-time timing constraints to be expressed, such as a requirement to read a given input within a stated time of it being produced.

```
box b in ( v :: int 32 ) out ( v' :: int 32 )
match x -> complex_fn x within 20ns
handle Timeout -> 0;
```

### 3.3 Device Declarations

Five kinds of device are supported: buffered streams (files), unbuffered FIFO streams, ports, memory-mapped devices, and interrupts. Each device has a directionality (for interrupts, this must be input), an operating system designator, and an optional time specification. The latter can be used to enable periodic scheduling or to ensure that critical events are not missed. Devices may be wired to box inputs or outputs. For example, we can define an operation to read a mouse periodically, returning true if the mouse button is down, as:

```
exception Timeout_mouseport;
port mouseport from "/dev/mouse" within 1ms raising Timeout_mouseport;

box readmouse in ( mousein :: bool ) out ( clicked :: bool )
match m -> m;
handle Timeout_mouseport -> false;

wire mouseport to readmouse.mousein;
wire readmouse.clicked to mouseuser.mouse;
```

### 3.4 Interrupt Handling

This section illustrates interrupt handling in Hume using an example adapted from [10], which shows how to write a simple parallel port device driver for Linux kernel version 2.4. Some simplifying assumptions have been made. Like many



other operating systems, Linux splits interrupt handlers into two halves: a *top-half* which runs entirely in kernel space, and which must execute in minimal time, and a *bottom-half* which may access user space memory, and which has more relaxed time constraints. For the parallel port, the top-half simply schedules the bottom-half for execution (using the `trig` wire), having first checked that the handler has been properly initialised. The IRQ and other device information are ignored.

```
box pp_int    -- Top half handler
in  ( (irq :: Int, dev_id :: string, regs :: Regs), active :: boolean )
out ( trig :: boolean, ki :: string )
match
  ( _, false ) -> ( *, "pp_int: spurious interrupt\n"),
  | ( _, true )  -> ( true, * ) ;

{-# kernel box pp_int -}
```

The bottom-half handler receives a time record produced by the top-half trigger output and the action generated by the parallel port. Based on the internal buffer state (a non-circular buffer represented as a byte array plus head and tail indexes), the internal state is modified and a byte will be read from/written to the parallel port. In either case, a log message is generated.

```
box pp_bottomhalf  -- Bottom half handler
in  ( td :: Time, buffer :: Buffer, action :: Acts )
out  ( buffer' :: Buffer, b :: Byte, log :: String )
match ( td, (head, tail, buf), Read b ) ->
  if tail < MAX_BUF then
    ( (head, tail+1, update buf tail b), *, log_read td )
  else ( (head, tail, buf), *, "buffer overrun" )
| ( td, (head, tail, buf), Write ) ->
  if head >= tail then ( 0, 0, "", * )
  else ( (head+1, tail, buf), buf @ head, *, log_write td );
```

We require an operation to trigger the bottom-half handler. This uses the standard `gettimeofday` Linux function to produce a time record. We also need to log outputs from `pp_int` using the kernel print routine `printk`. This is also specified as an operation. Finally we need to provide the parallel port interrupt routine with information about the IRQ to be used, the name of the handler etc.

The implementation of operations uses a foreign function interface based on that for Haskell [9] to attach a function call to a pseudo-box. The pseudo-box takes one input, marshals it according to the type of the arguments to the external call, executes the external call, and then unmarshals and returns its result (if any). In this way, we use the box mechanism to interface to impure foreign functions, and thereby ensure that box rules are purely functional without resorting to type-based schemes such as Haskell's monadic I/O.

```

operation trigger as "gettimeofday" :: Boolean -> Time;

operation kernel_info as "printk" :: String -> ();

interrupt parport from "(7,pp_int,0)"; -- assumes base at 0x378

wire parport to pp_int.info; wire initialised to pp_int.active;
wire pp_int.trig to trigger.inp; wire pp_int.ki to kernel_info.inp;

```

We now define a box `pp_do_write` to write the output value and the necessary two control bytes to strobe the parallel port. The control bytes are sequenced internally using a state transition, and output is performed only if the parallel port is not busy. We also define abstract ports (`pp_action`, ...) to manage the interaction with actual parallel port.

```

box pp_do_write      -- Write to the parallel port
in  ( cr, stat :: Byte, bout :: Byte, cr2 :: Byte )
out ( bout', cr' :: Byte, cr'' :: Byte )
match
  ( *, SP_SR_BUSY, *, *, * ) -> ( *, *, * ) -- wait until non-busy
| ( *, *, *, *, cr ) -> ( *, cr & ~SP_CR_STROBE, * )
| ( cr, _, bout, false, * ) -> ( bout, cr | SP_CR_STROBE, cr );

port pp_action; port pp_stat; port pp_data; port pp_ctl_in;
                                port pp_ctl_out;

```

Finally, wiring definitions link the bottom-half boxes with the parallel port.

```

wire trigger.outp to pp_bottomhalf.td;
wire pp_bottomhalf.buffer' to pp_bottomhalf.buffer;

wire pp_action to pp_bottomhalf.action;
wire pp_bottomhalf.b to bottomhalf.buffer;
wire pp_bottomhalf.log to kernel_info;

wire pp_ctl_in to pp_do_write.cr; wire pp_stat to pp_do_write.stat;
wire pp_do_write.bout to pp_data;
wire pp_do_write.cr' to pp_ctl_out;
wire pp_do_write.cr'' to pp_do_write.cr2;

wire pp_do_write.cr' to pp_do_write.cr2;

```

## 4 Modelling Space Costs

A major goal of the Hume project is to provide good space and time cost models for Hume programs. In the long term, we intend to provide cost models for all levels of Hume up to at least PR-Hume by exploiting emerging theoretical results concerning cost models for recursive programs [16]. We illustrate the practicality of our general approach by describing a simple space cost model for FSM-Hume

$$\boxed{
\begin{array}{c}
\frac{\text{space}}{E \vdash \text{exp} \Rightarrow \text{Cost}, \text{Cost}} \\
\frac{\text{space}}{E \vdash n \Rightarrow \mathcal{H}_{int32}, 1} \quad (1) \\
\ldots \\
\frac{E(\text{var}) = \langle h, s \rangle \quad \forall i. 1 \leq i \leq n, \quad \frac{\text{space}}{E \vdash \text{exp}_i \Rightarrow h_i, s_i}}{E \vdash \text{var exp}_1 \dots \text{exp}_n \Rightarrow \sum_{i=1}^n h_i + h, \max_{i=1}^n (s_i + (i-1)) + s} \quad (2) \\
\frac{\forall i. 1 \leq i \leq n, \quad \frac{\text{space}}{E \vdash \text{exp}_i \Rightarrow h_i, s_i}}{E \vdash \text{con exp}_1 \dots \text{exp}_n} \quad (3) \\
\Rightarrow \sum_{i=1}^n h_i + n + \mathcal{H}_{con}, \max_{i=1}^n (s_i + (i-1)) \\
\frac{\frac{\text{space}}{E \vdash \text{exp}_1 \Rightarrow h_1, s_1} \quad \frac{\text{space}}{E \vdash \text{exp}_2 \Rightarrow h_2, s_2} \quad \frac{\text{space}}{E \vdash \text{exp}_3 \Rightarrow h_3, s_3}}{E \vdash \text{if exp}_1 \text{ then exp}_2 \text{ else exp}_3} \quad (4) \\
\Rightarrow h_1 + \max(h_2, h_3), \max(s_1, s_2, s_3) \\
\frac{\frac{\text{decl}}{E \vdash \text{decls} \Rightarrow h_d, s_d, s'_d, E'} \quad \frac{\text{space}}{E' \vdash \text{exp} \Rightarrow h_e, s_e}}{E \vdash \text{let decls in exp} \Rightarrow h_d + h_e, \max(s_d, s'_d + s_e)} \quad (5)
\end{array}
}$$

**Fig. 3.** Space cost axioms for expressions

that predicts upper bound stack and space usage with respect to the prototype Hume Abstract Machine (**pHAM**) [13]. The stack and heap requirements for the boxes and wires represent the only dynamically variable memory requirements: all other memory costs can be fixed at compile-time based on the number of wires, boxes, functions and the sizes of static strings. In the absence of recursion, we can provide precise static memory bounds on rule evaluation. Predicting the stack and heap requirements for an FSM-Hume program thus provides complete static information about system memory requirements.

#### 4.1 Space Cost Rules

Figure 3 gives cost rules for a representative subset of FSM-Hume expressions, based on an operational interpretation of the **pHAM** implementation. The full set of cost rules is defined elsewhere [14]. We have shown that using this analysis, it is possible to construct programs that possess tightly bounded space behaviour

*in practice* for FSM-Hume. Heap and stack costs are each integer values of type `Cost`, labelled  $h$  and  $s$ , respectively. Each rule produces a pair of such values representing independent upper bounds on the stack and heap usage. The result is produced in the context of an environment,  $E$ , that maps function names to the heap and stack requirements associated with executing the body of the function, and which is derived from the top-level program declarations plus standard prelude definitions. Rules for building the environment are omitted here, except for local declarations, but can be trivially constructed.

The heap cost of a standard integer is given by  $\mathcal{H}_{int32}$  (rule 1), with other scalar values costed similarly. The cost of a function application is the cost of evaluating the body of the function plus the cost of each argument (rule 2). Each evaluated argument is pushed on the stack before the function is applied, and this must be taken into account when calculating the maximum stack usage. The cost of building a new data constructor value such as a user-defined constructed type (rule 3) is similar to a function application, except that pointers to the arguments must be stored in the newly created closure (one word per argument), and fixed costs  $\mathcal{H}_{con}$  are added to represent the costs of tag and size fields. The heap usage of a conditional (rule 4) is the heap required by the condition part plus the maximum heap used by either branch. The maximum stack requirement is simply the maximum required by the condition and either branch. Case expressions (omitted) are costed analogously. The cost of a `let`-expression (rule 5) is the space required to evaluate the value definitions (including the stack required to store the result of each new value definition) plus the cost of the enclosed expression. The local declarations are used to derive a quadruple comprising total heap usage, maximum stack required to evaluate any value definition, a count of the value definitions in the declaration sequence (used to calculate the size of the stack frame for the local declarations), and an environment mapping function names to heap and stack usage. The body of the `let`-expression is costed in the context of this extended environment.

## 5 Implementations of Hume

We have constructed two prototype implementations of Hume: a proof-of-concept reference interpreter written in the non-strict functional language, Haskell; and a basic abstract machine compiler [13], using the same Haskell front-end, and supported by a small runtime system written in C. We are in the process of producing a more sophisticated abstract machine implementation, and intend to produce fully compiled implementations in the near future. These implementations may be downloaded from <http://www.hume-lang.org>.

The prototype Hume abstract machine compiler (`phamc`) targets a novel abstract machine for Hume supporting concurrency, timeouts, asynchronicity, exceptions and other required constructs, the prototype Hume Abstract Machine (`pHAM`). The `pHAM` is primarily intended to provide a more realistic target for our work on cost modelling and analysis rather than as a production implementation. The compiler is defined using a formal translation scheme from Hume

source to abstract machine code. We have also provided an informal operational semantics for Hume abstract machine instructions. Our long-term intention is to use formal methods to verify the correctness of the compiler with respect to the Hume language semantics, and to verify the soundness of the source-level cost models with respect to an actual implementation. The **phamc** compiler has been ported to two Unix platforms: Red Hat Linux and Mac OS/X. We have also provided an implementation for the real-time operating system RTLinux [3]. The C back-end should prove highly portable to non-Unix platforms.

We have compared the performance of the **pHAM** against that of the most widely used abstract machine implementation: the Java Virtual Machine. A number of new embedded applications (predominantly in the mobile phone sector) are seriously targeting the Java software platform, so this is a fair comparison against an equivalent general purpose approach. Performance results [13] show that the performance of the **pHAM** implementation is consistently 9-12 times that of the standard JVM on various test-beds, including simple tests such as loop iteration. Compared with Sun's KVM for embedded systems [30], we thus estimate execution speed to be 12-20 times; and compared with the just-in-time KVM implementation [31], we estimate performance at 2-4 times. We conjecture that these performance gains arise from the use of a higher-level, more targeted abstract machine design. In the embedded systems domain, we do not need to include dynamic security checks, for example.

We have measured dynamic memory usage for a moderately complex control system [13] in the **pHAM** at less than 9KB of dynamic memory, and total memory usage, including static allocation, code, runtime system and operating system code at less than 62KB (more than half of this is required by the operating system – the **pHAM** implementation requires less than 32KB in total). These figures are approximately 50% of those for the corresponding KVM implementation, and suggest that the **pHAM** implementation should be suitable for fairly small-scale embedded control systems.

We are in the process of constructing a distributed implementation of Hume based on the **pHAM** implementation and mapping boxes to embedded system components. One outstanding technical issue is how to design the scheduling algorithm in such a way as to maintain consistency with the concurrent implementation that we have described here. We also intend to construct native code implementations and to target a wider range of hardware architectures. We do not foresee any serious technical difficulties with either of these activities.

## 6 Related Work

Accurate time and space cost-modelling is an area of known difficulty for functional language designs [28]. Hume is thus, as far as we are aware, unique in being based on strong automatic cost models, and in being designed to allow straightforward space- and time-bounded implementation for hard real-time systems. A number of functional languages have, however, looked at *soft* real-time issues (e.g. Erlang [2] or E-FRP [33], there has been work on using functional no-

tations for hardware design (essentially at the HW-Hume level) (e.g. Hydra [24]), and there has been much recent theoretical interest both in the problems associated with costing functional languages (e.g. [28, 20, 21]) and in bounding space/time usage (e.g. [34, 19]).

In a wider framework, two extreme approaches to real-time language design are exemplified by SPARK Ada [4] and the real-time specification for Java (RTSJ) [7]. The former epitomises the idea of language design by elimination of unwanted behaviour from a general-purpose language, including concurrency. The remaining behaviour is guaranteed by strong formal models. In contrast, the latter provides specialised runtime and library support for real-time systems work, but makes no absolute performance guarantees. Thus, SPARK Ada provides a minimal, highly controlled environment for real-time programming emphasising *correctness by construction* [1], whilst Real-Time Java provides a much more expressible, but less controlled environment, without formal guarantees. Our objective with the Hume design is to maintain correctness whilst providing high levels of expressibility.

## 6.1 Synchronous Dataflow Languages

In synchronous dataflow languages, unlike Hume, every *action* (whether computation or communication) has a zero time duration. In practice this means that actions must complete before the arrival of the next event to be processed. Communication with the outside world occurs by reacting to external stimuli and by instantaneously emitting responses. Several languages have applied this model to real-time systems control. For example, Signal [5] and Lustre [12] are similar declarative notations, built around the notion of timed sequences of values. Esterel [6] is an imperative notation that can be translated into finite state machines or hardware circuits, and Statecharts [17] uses a visual notation, primarily for design. One obvious deficiency is the lack of expressiveness, notably the absence of recursion and higher-order combinators. Synchronous Kahn networks [8] incorporate higher-order functions and recursion, but lose strong guarantees of resource boundedness.

## 6.2 Static Analyses for Bounding Space or Time Usage

There has been much recent interest in applying static analysis to issues of bounded time and space, but none is capable of dealing with higher-order, polymorphic and generally recursive function definitions as found in full Hume. For example, region types [34] allow memory cells to be tagged with an allocation *region*, whose scope can be determined statically. When the region is no longer required, all memory associated with that region may be freed without invoking a garbage collector. This is analogous to the use of Hume boxes to scope memory allocations. Hofmann’s linearly-typed functional programming language LFPL [18] uses linear types to determine resource usage patterns. First-order LFPL definitions can be computed in bounded space, even in the presence of

general recursion. For arbitrary higher-order functions, however, an unbounded stack is required.

Building on earlier work on sized types [20, 28], we have developed an automatic analysis to *infer* the *upper bounds* on evaluation costs for a simple, but representative, functional language with parametric polymorphism, higher-order functions and recursion [35]. Our approach assigns finite costs to a non-trivial subset of primitive recursive definitions, and is *automatic* in producing cost equations without any user intervention, even in the form of type annotations. Obtaining closed-form solutions to the costs of recursive definitions currently requires the use of an external recurrence solver, however.

## 7 Conclusions and Further Work

This chapter has introduced Hume, a domain-specific language for resource-limited systems such as the real-time embedded systems domain. The language is novel in being built on a combination of finite state machine and  $\lambda$ -calculus concepts. It is also novel in aiming to provide a high level of programming abstraction whilst maintaining good formal properties, including bounded time and space behaviour and provably correct rule-based translation. We achieve the combination of a high level of programming abstraction with strong properties included bounded time and space behaviour through synthesising recent advances in theoretical computer science into a coherent pragmatic framework. By taking a domain-specific approach to language design we have thus raised the level of programming abstraction without compromising the properties that are essential to the embedded systems application domain.

A number of important limitations remain to be addressed:

1. space and time cost models must be defined for additional Hume layers including higher-order functions and (primitive) recursion and these must be implemented as static analyses;
2. we need to provide machine-code implementations for a variety of architectures that are used in the real-time embedded systems domain, and to develop realistic demonstrator applications that will explore practical aspects of the Hume design and implementation;
3. more sophisticated scheduling algorithms could improve performance, however, these must be balanced with the need to maintain correctness;
4. no attempt is made to avoid deadlock situations through language constructs: a suitable model checker must be designed and implemented.

Of these, the most important limitation is the development and application of more sophisticated theoretical cost models. Our sized time type-and-effect system is already capable of inferring theoretical costs in terms of reduction steps for higher-order polymorphic definitions [28]. Adapting the system to infer heap, stack and time costs for Hume programs should be technically straightforward. Moreover, we have recently extended this theoretical system to cover primitive recursive definitions. Incorporating this analysis into Hume will go a long way towards our goal of achieving high level real-time programming.

## References

1. P. Amey, “Correctness by Construction: Better can also be Cheaper”, *CrossTalk: the Journal of Defense Software Engineering*, March 2002, pp. 24–28.
2. J. Armstrong, S.R. Virding, and M.C. Williams, *Concurrent Programming in Erlang*, Prentice-Hall, 1993.
3. M. Barabanov, *A Linux-based Real-Time Operating System*, M.S. Thesis, Dept. of Comp. Sci., New Mexico Institute of Mining and Technology, June 1997.
4. J. Barnes, *High Integrity Ada: the Spark Approach*, Addison-Wesley, 1997.
5. A. Benveniste and P.L. Guernic, “Synchronous Programming with Events and Relations: the Signal Language and its Semantics”, *Science of Computer Programming*, **16**, 1991, pp. 103–149.
6. G. Berry. “The Foundations of Esterel”, In *Proof, Language, and Interaction*. MIT Press, 2000.
7. G. Bollela et al. *The Real-Time Specification for Java*, Addison-Wesley, 2000.
8. P. Caspi and M. Pouzet. “Synchronous Kahn Networks”, *SIGPLAN Notices* 31(6):226–238, 1996.
9. M. Chakravarty (ed.), S.O. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S.L. Peyton Jones, A. Reid, M. Wallace and M. Weber, “The Haskell 98 Foreign Function Interface 1.0”, <http://www.cse.unsw.edu.au/~chak/haskell/ffi>, December, 2003.
10. J. Corbet and A. Rubini, “Linux Device Drivers”, 2nd Edition, O’Reilly, 2001.
11. The Ganssle Group. Perfecting the Art of Building Embedded Systems. <http://www.ganssle.com>, May 2003.
12. N. Halbwachs, D. Pilaud and F. Ouabdesselam, “Specifying, Programming and Verifying Real-Time Systems using a Synchronous Declarative Language”, in *Automatic Verification Methods for Finite State Systems*, J. Sifakis (ed.), Springer-Verlag, 1990, pp. 213–231.
13. K. Hammond. “An Abstract Machine Implementation for Embedded Systems Applications in Hume”, *Submitted to 2003 Workshop on Implementations of Functional Languages (IFL 2003)*, Edinburgh, 2003.
14. K. Hammond and G.J. Michaelson “Predictable Space Behaviour in FSM-Hume”, *Proc. 2002 Intl. Workshop on Impl. Functional Langs. (IFL ’02)*, Madrid, Spain, Springer-Verlag LNCS 2670, 2003.
15. K. Hammond and G.J. Michaelson, “Hume: a Domain-Specific Language for Real-Time Embedded Systems”, *Proc. Conf. on Generative Programming and Component Engineering (GPCE ’03)*, Springer-Verlag LNCS, 2003.
16. K. Hammond, H.-W. Loidl, A.J. Rebón Portillo and P. Vasconcelos, “A Type-and-Effect System for Determining Time and Space Bounds of Recursive Functional Programs”, *In Preparation*, 2003.
17. D. Harel, “Statecharts: a Visual Formalism for Complex Systems”, *Science of Computer Programming*, **8**, 1987, pp. 231–274.
18. M. Hofmann. A Type System for Bounded Space and Functional In-place Update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
19. M. Hofmann and S. Jost, “Static Prediction of Heap Space Usage for First-Order Functional Programs”, *Proc. POPL’03 — Symposium on Principles of Programming Languages*, New Orleans, LA, USA, January 2003. ACM Press.
20. R.J.M. Hughes, L. Pareto, and A. Sabry. “Proving the Correctness of Reactive Systems Using Sized Types”, *Proc. POPL’96 — ACM Symp. on Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.



21. R.J.M. Hughes and L. Pareto, "Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming", *Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99)*, aris, France, pp. 70–81, 1999.
22. S.D. Johnson, *Synthesis of Digital Designs from Recursive Equations*, MIT Press, 1984, ISBN 0-262-10029-0.
23. J. McDermid, "Engineering Safety-Critical Systems", I. Wand and R. Milner(eds), *Computing Tomorrow: Future Research Directions in Computer Science*, Cambridge University Press, 1996, pp. 217–245.
24. J.T. O'Donnell, "The Hydra Hardware Description Language", *This proc.*, 2003.
25. S.L. Peyton Jones (ed.), L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, K. Hammond, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, J.C. Peterson, A. Reid, and P.L. Wadler, *Report on the Non-Strict Functional Language, Haskell (Haskell98)* Yale University, 1999.
26. S.L. Peyton Jones, A.D. Gordon and S.O. Finne "Concurrent Haskell", *Proc. ACM Symp. on Princ. of Prog. Langs.*, St Petersburg Beach, Fl., Jan. 1996, pp. 295–308.
27. R. Pointon, "A Rate Analysis for Hume", *In preparation*, Heriot-Watt University, 2004.
28. A.J. Rebón Portillo, Kevin Hammond, H.-W. Loidl and P. Vasconcelos, "Automatic Size and Time Inference", *Proc. Intl. Workshop on Impl. of Functional Langs. (IFL 2002)*, Madrid, Spain, Sept. 2002, Springer-Verlag LNCS 2670, 2003.
29. M. Sakkinen. "The Darker Side of C++ Revisited", *Technical Report 1993-I-13*, <http://www.kcl.ac.uk/kis/support/cit/fortran/cpp/dark-cpl.ps>, 1993.
30. T. Sayeed, N. Shaylor and A. Taivalsaari, "Connected, Limited Device Configuration (CLDC) for the J2ME Platform and the K Virtual Machine (KVM)", *Proc. JavaOne – Sun's Worldwide 2000 Java Developers Conf.*, San Francisco, June 2000.
31. N. Shaylor, "A Just-In-Time Compiler for Memory Constrained Low-Power Devices", *Proc. 2nd Usenix Symposium on Java Virtual Machine Research and Technology (JVM '02)*, San Francisco, August 2002.
32. E. Schoitsch. "Embedded Systems – Introduction", *ERCIM News*, **52**:10–11, 2003.
33. W.Taha, "Event-Driven FRP", *Proc. ACM Symp. on Practical Applications of Declarative Languages (PADL '02)*, 2002.
34. M. Tofte and J.-P. Talpin, "Region-based Memory Management", *Information and Control*, **132**(2), 1997, pp. 109–176.
35. P. Vasconcelos and K. Hammond. "Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs", *Submitted to 2003 Workshop on Implementations of Functional Languages (IFL 2003)*, Edinburgh, 2003.

# Embedding a Hardware Description Language in Template Haskell

John T. O'Donnell

University of Glasgow, United Kingdom

**Abstract.** Hydra is a domain-specific language for designing digital circuits, which is implemented by embedding within Haskell. Many features required for hardware specification fit well within functional languages, leading in many cases to a perfect embedding. There are some situations, including netlist generation and software logic probes, where the DSL does not fit exactly within the host functional language. A new solution to these problems is based on program transformations performed automatically by metaprograms in Template Haskell.

## 1 Introduction

The development of a computer hardware description language, called Hydra, provides an interesting perspective on embedding as an implementation technique for domain specific languages (DSLs). Hydra has many features that fit smoothly within a higher order, nonstrict pure functional language, and these aspects of Hydra demonstrate the effectiveness and elegance of embedding. There are several areas, however, where Hydra does not fit perfectly within the host functional language. This creates technical difficulties that must be solved if the embedding is to work. Therefore an overview of the history of Hydra implementations provides a number of insights into embedded implementations of DSLs.

The Hydra project grew out of the work by Steven Johnson on using recursion equations over streams to describe the behavior of digital circuits [2]. Hydra was also inspired by Ruby [3], a relational language for circuit design. Hydra has been developed gradually over the last twenty years [4], [5], [6], [7], [8], concurrently with the development of functional languages. Hydra has been embedded in six related but distinct languages over the years: Daisy, Scheme, Miranda, LML, Haskell [1], and now Template Haskell [10].

The rest of this paper describes some of the main concepts in Hydra and how they were embedded in a functional language. The presentation follows a roughly chronological approach, showing how the implementation has gradually become more sophisticated. However, one major liberty will be taken with the history: Haskell notation will be used throughout the paper, even while discussing embedding techniques that were developed for earlier functional languages. Some of the earlier papers on Hydra, cited in the bibliography, use the old language notations.

Many of the features of Hydra fit precisely within a pure functional language, needing only a library of definitions of ordinary classes, instances, functions, types, and values. Let us call this a *perfect embedding*, because the target language Hydra fits perfectly within the features of the host language Haskell. Sections 2 and 3 describe the parts of Hydra that work as a perfect embedding. Sometimes, however, the domain specific language has a semantic inconsistency with the host language that interferes with the embedding. Section 4 presents such a situation, and shows how the mismatch has been solved using metaprogramming with Template Haskell.

## 2 A Perfect Embedding

The central concept of functional programming is the mathematical function, which takes an argument and produces a corresponding output. The central building block of digital circuits is the component, which does the same. This fundamental similarity lies at the heart of the Hydra embedding: a language that is good at defining and using functions is also likely to be good at defining and using digital circuits.

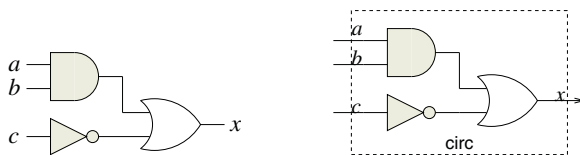
### 2.1 Circuits and Functions

Hydra models a component, or circuit, as a function. The arguments denote inputs to the circuit, and the result denotes the outputs. An equation is used to give a name to a wire (normally called a *signal*):

```
x = or2 (and2 a b) (inv c)
```

We can turn this circuit into a black box which can be used like any other component by defining a function (Figure 1).

```
circ a b c = x
  where x = or2 (and2 a b) (inv c)
```



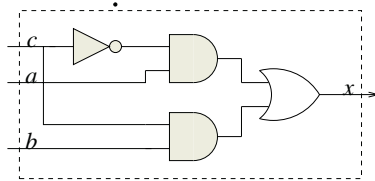
**Fig. 1.** Modeling a black box circuit as a function

In order to make the circuit executable, the basic logic gates need to be defined. A simple approach is to treat signals as booleans, and to define the logic gates as the corresponding boolean operators.

```
inv = not
and2 = (&&)
or2 = (||)
```

A useful building block circuit is the multiplexor `mux1`, whose output can be described as “if  $c = 0$  then  $a$  else  $b$ ” (Figure 2):

```
mux1 c a b =
  or2 (and2 (inv c) a) (and2 c b)
```



**Fig. 2.** The multiplexor circuit

## 2.2 Modeling State with Streams

The functional metaphor used above requires each circuit to act like a pure mathematical function, yet real circuits often contain state. A crucial technique for using streams (infinite lists) to model circuits with state was discovered by Steven Johnson [2]. The idea is to treat a signal as a sequence of values, one for each clock cycle. Instead of thinking of a signal as something that changes over time, it is a representation of the entire history of values on a wire. This approach is efficient because lazy evaluation and garbage collection combine to keep only the necessary information in memory at any time.

The delay flip flop `dff` is a primitive component with state; at all times it outputs the value of its state, and at each clock tick it overwrites its state with the value of its input. Let us assume that the flip flop is initialized to 0 when power is turned on; then the behavior of the component is defined simply as

```
dff x = False : x
```

Now we can define synchronous circuits with feedback. For example, the 1-bit register circuit has a load control `ld` and a data input `x`. It continuously outputs its state, and it updates its state with `x` at a clock tick if `ld` is true.

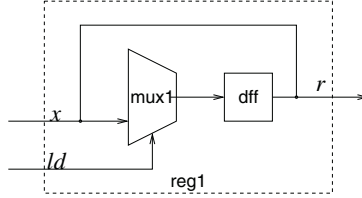
```
reg1 ld x = r
  where r = dff (mux1 ld r x)
```

The specification of `reg1` is natural and uncluttered, and it is also an executable Haskell program. The following test case defines the values of the input signals for several clock cycles, and performs the simulation:

```
sim_reg1 = reg1 ld x
  where ld = [True, False, False, True, False, False]
        x  = [True, False, False, False, False, False]
```

This is now executed using the interactive Haskell interpreter `ghci`, producing the correct output:

```
*Main> sim_reg1
[False,True,True,True,False,False,False]
```

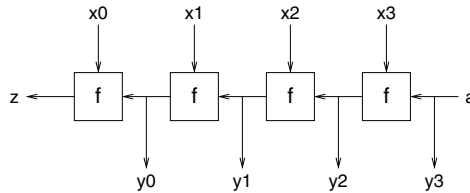


**Fig. 3.** Feedback in a synchronous register circuit

### 2.3 Design Patterns and Higher Order Functions

Digital circuits tend to have highly regular structures with enormous amounts of replication. These patterns are captured perfectly by higher order functions. For example, several important circuits have a structure (Figure 4) described by the `ascanr` function, one of an extremely useful family of map, fold and scan combinators.

```
ascanr :: (b->a->a) -> a -> [b] -> (a,[a])
```



**Fig. 4.** The `ascanr` pattern

A ripple carry adder can now be defined using `ascanr` to handle the carry propagation, and the `map2` combinator (similar to `zipWith`) to compute the sums.

```
add1 :: Signal a => a -> [(a,a)] -> (a,[a])
add1 c zs =
  let (c',cs) = ascanr bcarry c zs
      ss = map2 bsum zs cs
  in (c',ss)
```

This specification operates correctly on all word sizes. In other words, it defines the infinite class of  $n$ -bit adders, so the circuit designer doesn't need to design a 4-bit adder, and then an 8-bit one, and so on. Furthermore, we can reason formally about the circuit using equational reasoning. A formal derivation of an  $O(\log)$  time parallel adder, starting from this  $O(n)$  one, is presented in [9].

### 3 Alternative Semantics and Type Classes

An unusual aspect of Hydra is that a circuit specification has several semantics, not just one. This led to a mismatch between the target and host language which persisted for about ten years and through four host programming languages (Daisy, Scheme, Miranda and LML). The mismatch was finally resolved when type classes were introduced in Haskell.

#### 3.1 Hardware Models and Multiple Semantics

There is generally only one thing to be done with a computer program: run it. For this reason, we are accustomed to thinking about *the* denotational semantics of a program. In contrast, there are several tasks that must be performed with a circuit specification, including simulation, timing analysis, netlist generation, and layout. Furthermore, there are many different ways to simulate a circuit, corresponding to various models of hardware.

One way to handle the multiple semantics of a circuit specification is to introduce, for each semantics, a special representation for signals and corresponding implementations of the primitive components. These definitions are organized into modules, one for each semantics. A specification is a function definition containing free variables (`dff`, `and2`, etc.), which are resolved by loading up the module containing definitions for the desired semantics.

The early versions of Hydra used modules in exactly this way. To simulate a circuit, the designer loads a simulation module and the circuit specification module, and then simply executes the circuit. The other semantics are obtained just by loading the appropriate module.

This method worked adequately, but it did have some drawbacks. The main problem is that it forces the designer to work entirely within one semantic world at a time, yet it is quite natural to want to evaluate an expression according to one semantics in the course of working within another one.

#### 3.2 The Signal Class

Type classes in Haskell provide a more flexible, elegant and secure way of controlling the multiple circuit semantics. The `Signal` class defines a minimal set of operations that can be performed on all signals, regardless of their representation. For example, there are constant signals `zero` and `one`, and basic logic gates such as the inverter and 2-input and gate, for all signal types.

```
class Signal a where
  zero, one :: a
  inv :: a -> a
  and2 :: a -> a -> a
  ...
```

One semantics for combinational circuits uses the `Bool` type to represent signals:

```

instance Signal Bool where
  zero = False
  ...
  inv = not
  and2 a b = a && b
  or2 a b = a || b
  ...

```

The `Static` class defines additional methods that are meaningful only for the value of a signal at one time. For example, the `is0` and `is1` methods are used to give a Boolean interpretation to a signal.

```

class Signal a => Static a where
  is0, is1 :: a -> Bool
  showSigChar :: a -> Char
  readSigChar :: Char -> a
  ...

```

The static signal types include Booleans for basic simulations, and multiple valued types for representing CMOS signals. The details of the CMOS type are not important here; the point is that there are many representations of static signals.

```

instance Static Bool where
  is0 = not
  is1 = id
  ...

instance Static CMOS where
  is0 x = case x of
    Bot      -> False
    Weak0    -> True
    Weak1    -> False
    Strong0  -> True
    Strong1  -> False
    Top      -> False

```

There are other static signal instances that allow for richer circuit models, allowing techniques like tristate drivers, wired or, and bidirectional buses to be handled. A static signal can be lifted to a clocked one using streams:

```

instance Static a => Clocked (Stream a) where
  zero = Srepeat zero
  ...
  dff xs = Scons zero xs
  inv xs = Smap inv xs
  ...

```

Now we can execute a single circuit specification in different ways, simply by applying it to inputs of the right type:

```

circ :: Signal a => a -> a -> a -> a
circ a b c = x
  where x = or2 (and2 a b) (inv c)

test_circ_1 = circ False False False
test_circ_2 = circ
--
           0      1      2      3
           [False, False, True,  True]
           [False, True,  False, True]
           [False, True,  True,  True]
test_circ_3 = circ (Inport "a") (Inport "b") (Inport "c")
    
```

The following session with the Haskell interpreter `ghci` executes `circ` to perform a boolean simulation, a clocked boolean simulation, and a netlist generation (see Section 4).

```

*Main> test_circ_1
True
*Main> test_circ_2
[True,False,False,True]
*Main> test_circ_3
Or2 (And2 (Inport "a") (Inport "b")) (Inv (Inport "c"))
    
```

## 4 Preserving Referential Transparency

The preceding sections describe the implementation of Hydra in Haskell as a perfect embedding. The whole approach works smoothly because the foundation of functional programming – the mathematical function – is also the most suitable metaphor for digital circuits. Circuit specifications can be written in a style that is natural and concise, yet which is also a valid functional program.

In this section, we consider a crucial problem where the embedding is highly imperfect. (Another way of looking at it is that the embedding is *too* perfect, preventing us from performing some essential computations.) The problem is the generation of netlists; the state of the problem as of 1992 is described in [6]. A new solution, based on Template Haskell, will be presented in detail in a forthcoming paper, and some of the techniques used in the solution are described briefly in this section.

### 4.1 Netlists

A netlist is a precise and complete description of the structure of a circuit. It contains a list of all the components, and a list of all the wires. Netlists are unreadable by humans, but there are many automated systems that can



take a netlist and fabricate a physical circuit. In a sense, the whole point of designing a digital circuit is to obtain the netlist; this is the starting point of the manufacturing process.

A result of the perfect embedding is that the simulation of a Hydra specification is faithful to the underlying model of hardware. The behavioral semantics of the circuit is identical to the denotational semantics of the Haskell program that describes it.

Sometimes, however, we don't want the execution of a Hydra specification to be faithful to the circuit. Examples of this situation include handling errors in feedback loops, inserting logic probes into an existing circuit, and generating netlists. The first two issues will be discussed briefly, and the rest of this section will examine the problem of netlists in more detail.

In a digital circuit with feedback, the maximum clock speed is determined by the critical path. The simulation speed is also affected strongly by the critical path depth (and in a parallel Hydra simulator, the simulation speed would be roughly proportional to the critical path depth). If there is a purely combinational feedback loop, as the result of a design error, then the critical path depth is infinite. But a faithful simulation may not produce an error message; it may go into an infinite loop, faithfully simulating the circuit failing to converge. It is possible that such an error will be detected and reported by the Haskell runtime system as a "black hole error", but there is no guarantee of this – and such an error message gives little understanding of where the feedback error has occurred.

A good solution to this problem is a circuit analysis that detects and reports feedback errors. However, that requires the ability to traverse a netlist.

Circuit designers sometimes test a small scale circuit using a prototype board, with physical chips plugged into slots and physical wires connecting them. The circuit will typically have a modest number of inputs and outputs, and a far larger number of internal wires. A helpful instrument for debugging the design is the logic probe, which has a tip that can be placed in contact with any pin on any of the chips. The logic probe has light emitting diodes that indicate the state of the signal. This tool is the hardware analogue of inserting print statements into an imperative program, in order to find out what is going on inside.

Logic probes are not supported by the basic versions of Hydra described in the previous sections, because they are not faithful to the circuit, so it is impossible to implement them in a perfect embedding. The Hydra simulation computes exactly what the real circuit does, and in a real circuit there is no such thing as a logic probe. Section 4.6 shows how metaprogramming solves this problem, enabling us to have software logic probes while retaining the correct semantics of the circuit.

## 4.2 Netlist Semantics

Hydra generates netlists in two steps. First, a special instance of the signal class causes the execution of a circuit to produce a graph which is isomorphic to the

circuit. Second, a variety of software tools traverse the graph in order to generate the netlist, perform timing analyses, insert logic probes, and so on.

The idea is that a component with  $n$  inputs is represented by a graph node tagged with the name of the component, with  $n$  pointers to the sources of the component's input signals. There are two kinds of signal source:

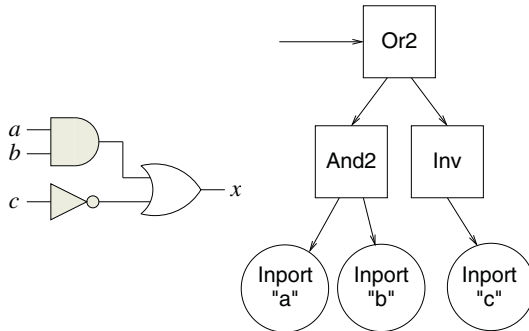
- An input to the circuit, represented as an **Inport** node with a **String** giving the input signal name.
- An output of some component, represented by a pointer to that component.

The following algebraic data type defines graphs where nodes correspond to primitive components, or inputs to the entire circuit. (This is not the representation actually used in Hydra, which is much more complex – all of the pieces of Hydra implementation given here are simplified versions, which omit many of the capabilities of the full system.)

```
data Net = Inport String | Dff Net | Inv Net
        | And2 Net Net | Or2 Net Net
```

The **Net** type can now be used to represent signals, so it is made an instance of the **Signal** class.

```
instance Signal Net where
  dff = Dff
  inv = inv
  and2 = And2
  or2 = Or2
```



**Fig. 5.** The schematic and graph of a combinational circuit

Figure 5 shows the graph of **circ** (defined in a previous section). This is constructed simply by applying the **circ** function to **Inport** nodes with signal names. Since **Inport**, applied to a string, returns a value of type **Net**, the entire execution of the circuit will select the **Net** instances for all signals, and the final output will be the circuit graph. This is illustrated by entering an application

of `circ` to input ports, using the interactive `ghci` Haskell system. The output is an `Or2` node, which points to the nodes that define the input signals to the `or2` gate.

```
*Main> circ (Inport "a") (Inport "b") (Inport "c")
Or2 (And2 (Inport "a") (Inport "b")) (Inv (Inport "c"))
```

Given a directed acyclic graph like the one in Figure 5, it is straightforward to write traversal functions that build the netlist, count numbers of components, and analyze the circuit in various ways.

### 4.3 Feedback and Equational Reasoning

When a circuit contains a feedback loop, its corresponding graph will be circular. Consider, for example, a trivial circuit with no inputs and one output, defined as follows:

```
oscillate = dff (inv oscillate)
```

Assuming that the flip flop is initialized to 0 when power is turned on, the circuit will oscillate between 0 and 1 forever. The Hydra simulation shows this:

```
*Main> oscillate
[False,True,False,True,False,True,False,True,False,True,False, ...]
```

Perhaps the deepest property of a pure functional language like Haskell (and Hydra) is referential transparency, which means that we can always replace either side of an equation by the other side. Now, in the equation

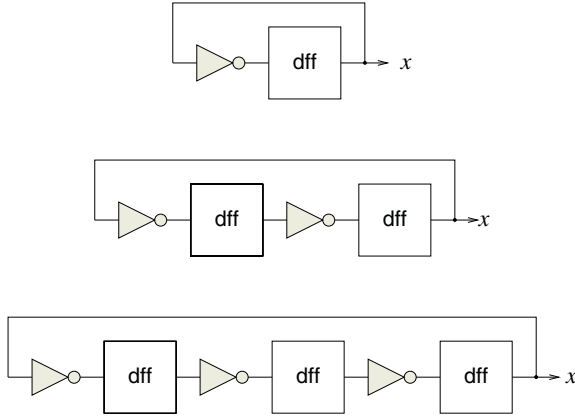
$$oscillate = dff \ (inv \ oscillate),$$

we can replace the *oscillate* in the right hand side by any value  $\alpha$ , as long as we have an equation  $oscillate = \alpha$ . And we do: the entire right hand side is equal to *oscillate*. The same reasoning can be repeated indefinitely:

```
oscillate = dff (inv oscillate)
          = dff (inv (dff (inv oscillate)))
          = dff (inv (dff (inv (dff (inv oscillate)))))
          ...
```

All of these circuits have exactly the same behavior. But it is less clear whether they have the same structure. A designer writing a specification with  $n$  flip flops might expect to end up with a circuit containing  $n$  flip flops, so the sequence of equations should arguably specify the sequence of circuits shown in Figure 6.

With existing Haskell compilers, the sequence of distinct equations above would indeed produce distinct graph structures. However, these graphs are circular, and there is no way for a pure Haskell program to traverse them without going into an infinite loop. Consequently, a Hydra tool written in Haskell cannot generate a netlist for a circuit with feedback written in the form above, and it cannot determine whether two of the graphs in Figure 6 are equal.



**Fig. 6.** The dff-inv circuit

#### 4.4 The Impure Solution: Pointer Equality

The first language in which Hydra was embedded was Daisy, a lazy functional language with dynamic types (like Lisp). Daisy was implemented by an interpreter that traversed a tree representation of the program, and the language offered constructors for building code that could then be executed. In short, Daisy supported metaprogramming, and this was exploited in early research on debugging tools, operating systems, and programming environments.

Daisy took the same approach as Lisp for performing comparisons of data structures: there was a primitive function that returned true if two objects are identical – that is, if the pointers representing them had the same value. This was used in turn by the standard equality predicate.

In the true spirit of embedding, the Daisy pointer equality predicate was used in the first version of Hydra to implement safe traversal of circuit graphs. This is a standard technique: the traversal function keeps a list of nodes it has visited before; when it is about to follow a pointer to a node, the traversal first checks to see whether this node has already been processed. This technique for netlist generation is described in detail in [5].

There is a serious and fundamental problem with the use of a pointer equality predicate in a functional language. Such a function violates referential transparency, and this in turn makes equational reasoning unsound.

The severity of this loss is profound: the greatest advantage of a pure functional language is surely the soundness of equational reasoning, which simplifies the use of formal methods sufficiently to make them practical for problems of real significance and complexity. For example, equational reasoning in Hydra can be used to derive a subtle and efficient parallel adder circuit [9].

The choice to use pointer equality in the first version of Hydra was not an oversight; the drawbacks of this approach were well understood from the outset, but it was also clear that netlist generation was a serious problem that would

require further work. The simple embedding with pointer equality provided a convenient temporary solution until a better approach could be found. The unsafe pointer equality method was abandoned in Hydra around 1990, and replaced by the labeling transformation discussed in the next section.

## 4.5 A Labeling Transformation

It is absolutely essential for a hardware description language to be able to generate netlists. There are only two possible approaches: either we must use impure language features, such as the pointer equality predicate, or we must find a way to write circuit specifications in a form that enables us to work around the problem.

The fundamental difficulty is that we need a way to identify uniquely at least one node in every feedback loop, so that the graph traversal algorithms can determine whether a node has been seen before. This can be achieved by decorating the circuit specification with an explicit labeling function:

```
label :: Signal a => Int -> a -> a
```

Now labels can be introduced into a circuit specification; for example, a labeled version of the `reg1` circuit might be written as follows:

```
reg1' ld x = r
  where r = label 100 (dff (mux1 ld r x))
```

It is straightforward to add `label` to the `Signal` class, and to define an instance of it which ignores the label for behavioral signal types:

```
instance Signal Bool where
  ...
  label s x = x
```

The labeled circuit `reg'` can be simulated just like the previous version:

```
sim_reg' = reg1' ld x
  where ld = [True, False, False, True, False, False]
        x  = [True, False, False, False, False, False]
```

In order to insert the labels into circuit graphs, a new `Label` node needs to be defined, and the instance of the label function for the `Net` type uses the `Label` constructor to create the node:

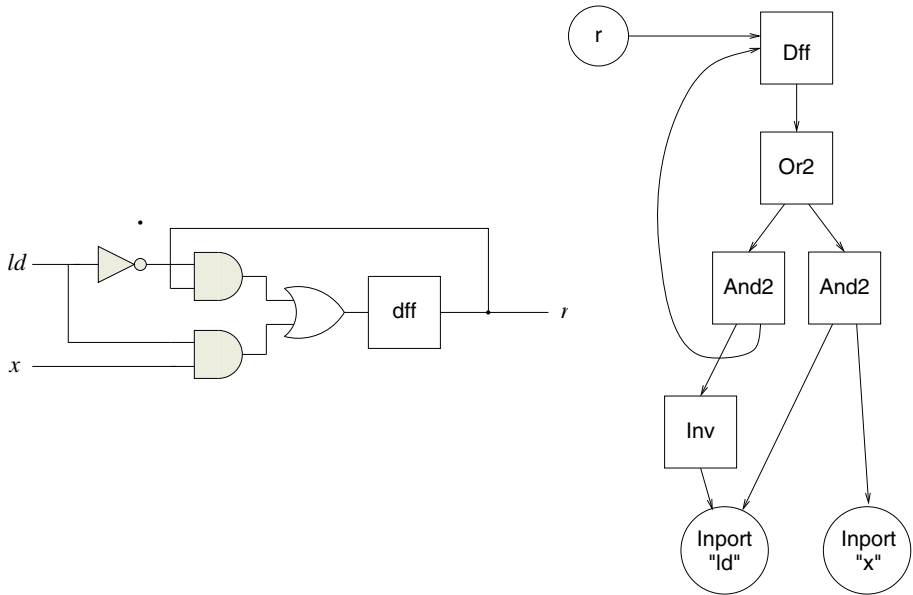
```
data Net = ...
  | Label Int Net

instance Signal Net where
  ...
  label = Label
```

Now we can define executions of the labeled specification for both simulation and netlist types:

```
sim_reg' = reg1' ld x
  where ld = [True, False, False, True, False, False]
        x  = [True, False, False, False, False, False]

graph_reg' = reg1' (Inport "ld") (Inport "x")
```



**Fig. 7.** Schematic and netlist for `reg1`

Figure 7 shows the schematic diagram and the corresponding circuit graph. The results of executing these test cases using `ghci` are shown below. The simulation produces the correct outputs. The graph can be printed out, but it is a circular graph so the output will be infinite (and is interrupted interactively by typing Control-C).

```
*Main> sim_reg'
[False,True,True,True,False,False,False]
*Main> graph_reg'
Label 100 (Dff (Or2 (And2 (Inv (Inport "ld")) (Label 100 (Dff (Or2
(And2 (Inv (Inport "ld")) (Label 100 (Dff (Or2 (And2 (Inv (Inport
"ld")) (Label 100 (DfInterrupted.
```

Although it is not useful to print the graph directly, the presence of labels makes it possible to traverse it in order to build a netlist. Rather than show a full

netlist traversal here, consider the equivalent but simpler problem of counting the number of flip flops in a circuit. This is achieved by the following function (there is an equation corresponding to each constructor in the `Net` type, but only a few are shown here):

```
count_dff :: [Int] -> Net -> Int
count_dff ls (Inport s) = 0
...
count_dff ls (Dff x) = 1 + count_dff ls x
...
count_dff ls (And2 x y) = count_dff ls x + count_dff ls y
...
count_dff ls (Label p x) =
  if p 'elem' ls
  then 0
  else count_dff (p:ls) x
```

Using labels, the sequence of circuits shown in Figure 6 can be defined unambiguously:

```
circloop, circloop1, circloop2, circloop3 :: Net
circloop = x
  where x = dff (inv x)
circloop1 = x
  where x = Label 1 (dff (inv x))
circloop2 = x
  where x = Label 1 (dff (inv (Label 2 (dff (inv x)))))
circloop3 = x
  where x = Label 1 (dff (inv (Label 2
    (dff (inv (Label 3 (dff (inv x))))))))
```

Executing the test cases produces the following results. First, the number of flip flops in the labeled register circuit is counted correctly. The unlabeled circuit results in a runtime exception, since the circuit graph is equivalent to an infinitely deep directed acyclic graph. The labeled circuits are all distinguished correctly.

```
*Main> count_dff [] graph_reg'
1
*Main> count_dff [] circloop
*** Exception: stack overflow
*Main> count_dff [] circloop1
1
*Main> count_dff [] circloop2
2
*Main> count_dff [] circloop3
3
```

The use of labeling solves the problem of traversing circuit graphs, at the cost of introducing two new problems. It forces a notational burden onto the circuit designer which has nothing to do with the hardware, but is merely an artifact of the embedding technique. Even worse, the labeling must be done correctly and yet cannot be checked by the traversal algorithms.

Suppose that a specification contains two different components that were mistakenly given the same label. Simulation will not bring out this error, but the netlist will actually describe a different circuit than the one that was simulated. Later on the circuit will be fabricated using the erroneous netlist. No amount of simulation or formal methods will help if the circuit that is built doesn't match the one that was designed.

When monads were introduced into Haskell, it was immediately apparent that they had the potential to solve the labeling problem for Hydra. Monads are often used to automate the passing of state from one computation to the next, while avoiding the naming errors that are rife with ordinary `let` bindings. Thus a circuit specification might be written in a form something like the following:

```
circ a b =  
  do p <- dff a  
    q <- dff b  
    x <- and2 p q  
  return x
```

The monad would be defined so that a unique label is generated for each operation; this is enough to guarantee that all feedback loops can be handled correctly.

However, there are two disadvantages of using monads for labeling in Hydra. The first problem is that monads introduce new names one at a time, in a sequence of nested scopes, while Hydra requires the labels to come into scope recursively, all at once, so that they are all visible throughout the scope of a circuit definition. In the above example, the definition of `p` cannot use the signal `q`.

A more severe problem is that the circuit specification is no longer a system of simultaneous equations, which can be manipulated formally just by “substituting equals for equals”. Instead, the specification is now a sequence of computations that – when executed – will yield the desired circuit. It feels like writing an imperative program to draw a circuit, instead of defining the circuit directly. Equational reasoning would still be sound with the monadic approach, but it would be far more difficult to use: the monadic circuit specification above contains no equations at all, and if the monadic operations are expanded out to their equational form, the specification becomes bloated, making it hard to manipulate formally.

The monadic approach offers a way to overcome the mismatch between the semantics of Hydra (which needs netlists) and Haskell (which disallows impure features). However, this is a case where a compromise to the DSL in order to make it fit within a general purpose language is not worthwhile; the loss of easy equational reasoning is too great a drawback, so the monadic approach was rejected for Hydra.



## 4.6 Hydra in Template Haskell

Instead of requiring the designer to insert labels by hand, or using monads, the labels could be inserted automatically by a program transformation. If we were restricted to the standard Haskell language, this would entail developing a parser for Hydra, an algebraic data type for representing the language, and functions to analyze the source and generate the target Haskell code, which can then be compiled. A project along these lines was actually underway when Template Haskell became available, offering a much more attractive approach.

Template Haskell [10] provides the ability for a Haskell program to perform computations at compile time which generate new code that can then be spliced into the program. It is similar in many ways to macros in Scheme, which have long been used for implementing domain specific languages within a small and powerful host.

Template Haskell defines a standard algebraic data type for representing the abstract syntax of Haskell programs, and a set of monadic operations for constructing programs. These are expressible in pure Haskell. Two new syntactic constructs are also introduced: a reification construct that gives the representation of a fragment of code, and a splicing construct that takes a code representation tree and effectively inserts it into a program.

The following definition uses the reification brackets `[d| ... |]` to define `circ_defs_rep` as an algebraic data type representing the code for a definition:

```
circ_defs_rep = [d|
  reg1 :: Clocked a => a -> a -> a
  reg1 ld x = r
    where r = dff (mux1 ld r x)
|]
```

The value of `circ_defs_rep` is a list of abstract syntax trees for the definitions within the brackets; this list contains a representation of the type declaration and the equation defining `reg1`. All the syntactic constructs, including the top level equation, the **where** clause, the equations within the **where** clause, all the way down to individual applications, literals, and variables, are represented as nodes in the abstract syntax tree.

Template Haskell is a homogeneous multistage language; that is, all the aspects of Haskell which the ordinary programmer can use are also available to process the abstract syntax tree at program generation time. Thus the Hydra `transform_module` function, which transforms the source code into the form that will be executed, is just an ordinary Haskell function definition.

The transformation used by Hydra is a generalization of the labeling transformation (Section 4.5), but the details of the representation are more complex than described there. The transformed circuit specification contains detailed information about the form of the specification itself, represented in data structures that can be used by software tools at runtime. For example, logic probes can be implemented in software, because simulation time functions can probe the data

structures generated by the transformed code in order to find any signal the user wants to see.

The `transform_module` function analyzes the code, checks for a variety of errors and issues domain-specific error messages if necessary, inserts the labels correctly, generates code that will build useful data structures for simulation time, and also performs some other useful tasks. The result of this is a new code tree. The `$(...)` syntax then splices the new code into the program, and resumes the compilation.

```
$(transform_module circ_defs_rep)
```

It would be possible to ask the circuit designer using Hydra to write the final transformed code directly, bypassing the need for metaprogramming. In principle, the effect of this automated transformation is the same as if the programmer had written the final code in the first place. In practice, however, there are some significant differences:

- The automated transformation guarantees to perform the transformation correctly. If the circuit designer were to insert labels incorrectly, the circuit netlist would not conform to the circuit being simulated. In other words, the designer could simulate the circuit, conclude that it is working, then fabricate the circuit, and find that the real hardware behaves differently than predicted by the simulation.
- The transformed code is verbose compared with the original Hydra circuit specification.
- As the Hydra system evolves, the transformation can be updated silently to provide whatever new capabilities are found necessary, without requiring all the Hydra users to rewrite all their existing circuit descriptions.

The algebraic data type used to represent the circuit graph contains more information than the simple `Net` type used earlier in this paper; in addition to supporting netlist generation, it also provides the information needed for several other software tools, including software logic probes. The transformation proceeds in the following main steps:

1. Use pattern matching to determine the kind of declaration.
2. Traverse the argument patterns to discover the names, and construct a renaming table.
3. Build new patterns incorporating the alpha conversions.
4. Copy the expressions with alpha conversions.
5. Construct the black box graph structure.
6. Generate unique labels and attach them to the nodes denoting the local equations.
7. Assemble the final transformed function definition.
8. Print out the original and transformed code, as well as temporary values, if requested by the user.

A full understanding of the Hydra program transformation is beyond the scope of this paper, but discussion of a few extracts from it may help to clarify how program transformations can be implemented in Template Haskell.

One task that arises in the transformation is to traverse an expression that occurs in a circuit definition, locate the variables that occur within it, create fresh variables to use in their place, and build a renaming table that gives the correspondences among the variables. This is performed by `findvars_exp`, which performs a case analysis on the expression. Here is part of the definition:

```
findvars_exp :: [Renaming] -> (ClusterLoc->ClusterLoc)
              -> Exp -> Q [Renaming]
findvars_exp rs f (Con s) = return rs
findvars_exp rs f (Var s) =
  do s1 <- gensym s
     let loc = f (CluVar s)
     return ((s,loc,s1,s1) : rs)
```

The first case handles constants, which are represented as a node `Con s`. Since no variables have been found in this case, the existing renaming table `rs` is returned. The next case, an expression `Var s`, represents a variable whose name is the string value of `s`. The function performs a monadic operation `gensym s` to create a fresh name, which will contain the string `s` along with a counter value. This fresh name is bound to `s1`. An entry for the renaming table is calculated, with the help of a functional argument `f`, and the result is attached to the head of the `rs` list. Many parts of the Hydra transformation involve similar recursions over abstract syntax trees in order to obtain the information needed to generate the final code.

The renaming table constructed by `findvars_exp` is used in a later stage of the transformation to build a new expression. This is performed by `update_exp`; part of its definition is:

```
update_exp :: [Renaming] -> (Renaming->String) -> Exp -> Exp
update_exp rs f (Con s)      = Con s
update_exp rs f (Var s)      = Var (ren_lookup f s rs)
```

These fragments of the transformation are typical: the program generation consists almost entirely of ordinary processing of trees expressed as algebraic data types. The monadic form of `findvars_exp` makes it convenient to perform a `gensym`, and there are other useful operations (such as input/output) which also require the transformation to be expressed monadically.

During a transformation, the metaprogram prints a detailed trace of what it is doing. Here is part of the output produced when the circuit `reg1` is transformed, showing the patterns and expressions that are discovered in the source code, and some of the variable renamings to be used:

```

*** Transforming declaration: Fun reg1
InPort patterns:
  1. Pvar "ld'75"
  2. Pvar "x'76"
Output expression:
  Var "r'77"
Left hand sides of the equations:
  0. Pvar "r'77"
Right hand sides of the equations:
  0. App (Var "Signal:dff") (App (App (App (Var "mux1")
    (Var "ld'75")) (Var "r'77")) (Var "x'76"))
Renamings for inputs:
  0. ("x'76",InportArg 1 (CluVar "x'76"),"x'76'283","x'76'284")
  1. ("ld'75",InportArg 0 (CluVar "ld'75"),
    "ld'75'281","ld'75'282")

```

The final transformed code is shown below. The unusual variable names, such as `r'77'286`, result from multiple uses of `gensym`. All the elements of the original definition are still discernible. The function is still defined as a top level equation with a **where** clause. Furthermore, the equation `r = dff (mux1 ld r x)` that appears in the **where** clause of the source code is still present, although it has been transformed to `r'77'285 = Signal.dff mux1 ld'75'282 r'77'286 x'76'284`. Additional equations have been added by the transformation. Some of these provide alternative names for signals with labels added, and the last equation defines a data structure `box` which contains a complete description of the structure of the circuit. The names used by the circuit designer (`reg1`, `ld`, `x`, `r`) all appear as strings in the data structures; this is needed so that a software logic probe, given the name of a signal, can locate it deep within the circuit.

```

reg1 :: forall a'74. (Signal.Clocked a'74) => a'74 -> a'74 -> a'74
reg1 ld'75'281 x'76'283 = r'77'286
  where
    r'77'285 = Signal.dff mux1 ld'75'282 r'77'286 x'76'284
    r'77'286 = alias r'77'285 "r" box (Equation 0 (CluVar "r"))
    ld'75'282 = alias ld'75'281 "ld" box
              (InportArg 0 (CluVar "ld"))
    x'76'284 = alias x'76'283 "x" box (InportArg 1 (CluVar "x"))
    box = Transform.mkbox ['r', 'e', 'g', '1']
          [ld'75'282, x'76'284]
          [r'77'286]
          [r'77'286]

```

The `alias` function is a method in the `Signal` class, since its definition depends on the mode of execution. If the definition is being executed to produce a netlist, then the structural instance of `alias` builds a graph node describing the signal. The new definition of `reg1` can also be simulated; this requires a method for `alias` which extracts the underlying signal value.

## 5 Conclusion

The design and implementation of Hydra is an example of embedding a domain specific language in a general purpose language. The embedding has always been fundamental to the whole conception of Hydra. At the beginning, in 1982, the project was viewed as an experiment to see how expressive functional languages really are, as well as a practical tool for studying digital circuits. It took several years before Hydra came to be seen as a domain specific language, rather than just a style for using functional languages to model hardware.

There are several criteria for judging whether an embedding is successful:

- *Does the host language provide all the capabilities needed for the domain specific language?* It's no good if essential services – such as netlist generation – must be abandoned in order to make the DSL fit within the host language.
- *Does the host language provide undesirable characteristics that would be inherited by the domain specific one?* This criterion is arguably a compelling reason not to use most imperative programming languages for hosting a hardware description language, because the sequential assignment statement has fundamentally different semantics than the state changes in a synchronous circuit. But users of VHDL would probably disagree with that assessment, and there is always a degree of opinion in assessing this criterion. For example, many people find the punctuation-free syntax of Haskell, with the layout rule, to be clean, readable, and elegant. Others consider it to be an undesirable feature of Haskell which has been inherited by Hydra.
- *Have the relevant advantages of the host language been retained?* The most uncompromising requirement of Hydra has always been support for equational reasoning, in order to make formal methods helpful during the design process. This is a good reason for using a functional language like Haskell, but it is also essential to avoid any techniques that make the formal reasoning unsound, or unnecessarily difficult.
- *Does the embedding provide a notation that is natural for the problem domain?* A domain specific language really must use notation that is suitable for the application domain. It is unacceptable to force users into a completely foreign notation, simply in order to save the DSL implementor the effort of writing a new compiler. Before Template Haskell became available, Hydra failed badly on this criterion. If the hardware designer is forced to insert labels manually, in order to solve the netlist problem, one could argue that the embedding is causing serious problems and a special purpose compiler would be more appropriate.

Based on these criteria, the embedding of Hydra in Template Haskell has been successful. All the necessary features of Hydra – concise specification, simulation and other software tools, software logic probes, netlist generation – are provided through embedding, by using Haskell's features. Arguably, an undesirable characteristic of Haskell that might be inherited by Hydra is the lack of a pointer equality predicate. However, we have avoided the need for pointer equality through metaprogramming – and this has enabled us to retain the crown-

ing advantage of Haskell, equational reasoning, which would have been lost if pointer equality were used! Finally, the notation for writing circuits is concise and lightweight; experience has shown that beginners are able to learn and use Hydra effectively to design digital circuits, even if they do not know Haskell.

The criteria above have been used as guiding principles through the entire project. Of course they are not simple binary success/failure measures. It is always fair to ask, for example, whether the notation could be made more friendly to a hardware designer if we did not need to employ the syntax of Haskell, and opinions differ as to what notations are the best. This situation is not peculiar to Hydra: all other hardware description languages also inherit syntactic clutter from existing programming languages, even when they are just inspired by the languages and are not embedded in them.

It is interesting to observe that most of the embedding of Hydra in Haskell does *not* rely on metaprogramming; it is almost a perfect embedding in standard Haskell. Although metaprogramming is not used throughout the entire implementation of Hydra, it is absolutely essential in the one place where it is used.

There is a continuum of possible embeddings of a DSL in a host language. At one extreme, a perfect embedding treats the DSL as just a library to be used with the host language, and no metaprogramming is needed. At the other extreme, the host language is used to write a conventional compiler, and the DSL inherits nothing directly from the host. Metaprogramming is also unnecessary here; one just needs a language suitable for compiler construction. Metaprogramming is most valuable for intermediate situations where the DSL can almost be implemented in the host language with a library of ordinary function definitions, but there is some mismatch between the semantics of the languages that causes it not quite to work. Hydra illustrates exactly that situation.

Template Haskell offers a significant improvement to the ability of Haskell to host domain specific languages. Wherever there is a mismatch between the DSL and Haskell, a metaprogram has the opportunity to analyze the source and translate it into equivalent Haskell; there is no longer a need to make the original DSL code serve also as the executable Haskell code.

In effect, Template Haskell allows the designer of a domain specific language to find the right mixture of embedding and compilation. It is no longer necessary for an embedding to be perfect in order to be useful. Haskell has excellent facilities for general purpose programming, as well as excellent properties for formal reasoning. Now that the constraints on embedding have been relaxed, Haskell is likely to find much wider use as the host for domain specific languages.

## References

1. Simon Peyton Jones (ed.). Haskell 98 language and libraries. *Journal of Functional Programming*, 13(1):1–255, January 2003.
2. Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, 1984. The ACM Distinguished Dissertation Series.

3. Geraint Jones and Mary Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods in VLSI Design*, chapter 1, pages 13–70. North-Holland, 1990. IFIP WG 10.5 Lecture Notes.
4. John O'Donnell. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382, Amsterdam, April 1987. North-Holland.
5. John O'Donnell. Hydra: hardware description in a functional language using recursion equations and high order combining forms. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 309–328, Amsterdam, 1988. North-Holland.
6. John O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 178–194. Springer-Verlag, 1992.
7. John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *FPLE'95: Symposium on Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 195–214. Springer-Verlag, 1995.
8. John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings 16th International Parallel & Distributed Processing Symposium*, page 234 (abstract). IEEE Computer Society, April 2002. Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications—PDSECA.
9. John O'Donnell and Gudula Rünger. Derivation of a carry lookahead addition circuit. *Electronic Notes in Theoretical Computer Science*, 59(2), September 2001. Proceedings ACM SIGPLAN Haskell Workshop (HW 2001).
10. Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

# A DSL Paradigm for Domains of Services: A Study of Communication Services

Charles Consel and Laurent Réveillère

INRIA – LaBRI

ENSEIRB – 1, avenue du docteur Albert Schweitzer

Domaine universitaire - BP 99

F-33402 Talence Cedex, France

{consel,reveillere}@labri.fr

<http://compose.labri.fr>

**Abstract.** The domain of services for mobile communication terminals has long become a fast-moving target. Indeed, this domain has been affected by a continuous stream of technological advances on aspects ranging from physical infrastructures to mobile terminals. As a result, services for this domain are known to be very unpredictable and volatile. This situation is even worse when considering services relying heavily on multimedia activities (*e.g.*, games, audio and/or video messages, etc.). Such an application area is very sensitive to a large variety of aspects such as terminal capabilities (graphics, CPU, etc.), bandwidth, service provider's billing policies, QoS, and user expectations.

To address these issues, we present a paradigm based on domain-specific languages (DSLs) that enables networking and telecommunication experts to quickly develop robust communication services. Importantly, we propose implementation strategies to enable this paradigm to be supported by existing software infrastructures.

Our DSL paradigm is uniformly used to develop a platform for communication services, named Nova. This platform addresses various domains of services including telephony services, e-mail processing, remote-document processing, stream processing, and HTTP resource adaption.

## 1 Introduction

Recent technological advances in physical infrastructures and terminals make it possible to offer a vast variety of communication services. In particular, many wireless technologies, like GPRS, UMTS, 802.11 and Bluetooth, ensure sufficient bandwidth for multimedia applications dealing with audio and video streams [1, 2]. Furthermore, handheld computers are getting smaller and smaller and are beginning to offer performance competing with low-end PCs in terms of graphical capabilities and computing power. Also, handheld computers and telephone terminals are converging, as suggested by the number of applications they both offer (schedules, address books, note books, etc.). These trends should rapidly turn these two devices into one.



In fact, looking at the future of telecommunications, the question is no longer when customers will have a communication terminal with enough bandwidth to run multimedia applications. As reported by several major telecommunication players [3], this situation will occur in the near future with UMTS terminals, or comparable technologies. Rather, the critical question is: *What services should a communication terminal offer?*

Not only is this question not answered today, but, as in any emerging area, the needs and expectations of customers are unpredictable and volatile.

*Services are unpredictable* because communication terminals offer possibilities that average customers do not actually grasp. In fact, the logic that dictates the dynamics of the market is not based on technical considerations. For example, SMS-based services represent a rather unexpected booming sector considering that typing messages on the primitive keyboard of a GSM phone is a notoriously laborious process. Yet, SMS messages have proved to be a lasting fad generating sizable revenues.

*Services are volatile* because the capabilities of communication terminals enable an endless set of potential services; to a large extent, human creativity is the limit in creating new services! As a consequence, offering a fixed set of services is a limiting factor to the dissemination of the technology. In fact, like services for Intelligent Networks, time-to-market will surely be a key competitive advantage when tracking customer needs for new services.

Unpredictability and volatility make the supply of new services vital. In the context of telecommunications, the platform owners should not constrain third-party service providers. Instead, they should encourage them to participate in the service creation process so as to increase and diversify the supply of new services. Currently, platforms are often closed to potential service providers because conventional models rely on controlling the market from the platform to the end-users. Beyond economical reasons, platform openness is prohibited because it compromises the robustness of the platform.

## Our Approach

To address the key issues of service creation, we introduce a paradigm based on domain-specific languages (DSLs). A DSL is developed for a domain of services. It provides networking and telecommunication experts with dedicated syntax and semantics to quickly develop robust variations of services within a particular domain. Because of a critical need for standardization, the networking and telecommunication area critically relies on a protocol specification to define a *domain of services*. Our approach aims to introduce variations in a domain of services without requiring changes to the protocol. Furthermore, the DSL is restricted so as to control the programmability of variations. Finally, we describe how the most common software architecture of the networking and telecommunication area, namely the *client-server model*, can be extended to support our DSL paradigm.

We use our DSL paradigm to develop the Nova platform that consists of different domains of services, namely, telephony services, e-mail processing, remote-document processing, stream and HTTP resource adapters. A DSL has been developed for each domain of services; the definition of this domain of services is itself derived from a specific protocol.

To illustrate our presentation, we consider services dedicated to access mailboxes remotely. Such services allow an end-user to access messages stored on a remote server. In fact, there are many ways in which these services can be realized. One domain of services is defined by the Internet Message Access Protocol (IMAP) [4, 5]. We show that variations need to be introduced in this domain of services to adapt to a variety of user requirements. We develop a DSL to enable service variation to be introduced without compromising the robustness of the e-mail server.

## Overview

Section 2 identifies the key requirements of a platform for communication services. Section 3 describes how to introduce and control programmability in such a platform. Section 4 presents strategies to introduce programmability in the client-server model. Section 5 shows how it scales up to a complete platform for communication services. Section 6 discusses lessons learned during the development of Nova. Section 7 gives concluding remarks.

## 2 Requirements

In this section, we present the key requirements that a platform should fulfill to successfully address the rapidly evolving domain of communication services.

**Openness.** Most families of services are bound to evolve, often rapidly and unpredictably in emerging domains such as multimedia communications. To address this key issue, a platform for communication services has to be open. This openness should enable a wide variety of services to be easily introduced. In fact, each community of users ought to be offered specific services, applications and ways of communicating that reflect their interests, cultural backgrounds, social codes, etc.

**Robustness.** An open platform for communication services is destined to be shared and fueled by as many service providers as possible. As in the context of software providers, service providers can either be companies, or individuals devoted to some community, as illustrated by the development effort of Linux. Like software, new services can only spread without restrictions if safety and security are guaranteed. For example, in the context of telephony services, a buggy or malicious call-processing service may crash (or compromise) the entire underlying signaling platform.

**Composability.** When developing a software system, a programmer typically delegates some treatments to specific software components. Similarly, when developing a sophisticated service, one would want to combine some sub-families of services. As a consequence, different services should not interfere and should be able to be combined and used intensively without degrading the performance of the platform.

**Scalability.** Our target platform should be scalable with respect to various aspects. It should be open enough to cover most emerging user needs; its robustness should be demonstrated in the context of a sizable community of service providers, and address the critical properties of each domain of services; finally, the platform should offer appropriate support for a wide range of communication services.

### 3 The DSL Paradigm

In this section, we present the main aspects of our DSL paradigm, starting from a protocol and ending with a DSL.

#### 3.1 From a Protocol to a Domain of Services

A protocol goes beyond the definition of the rules and conventions used by a server and a client to interact. A protocol is the outcome of a careful analysis of a domain of services. It introduces the fundamental abstractions of a target domain of services in the form of requests. These abstractions are parameterized with respect to specific client data. These parameters and the server responses suggest key data types for the domain of services. Examining a protocol to determine its domain of services is a valuable approach because communication services in networking and telecommunications are systematically based on a protocol.

For example, the Internet Message Access Protocol (IMAP) defines a domain of services for remote access to mailboxes [4, 5]. This domain of services aims to provide a user with access to messages stored on a remote server.

A protocol for communication services is traditionally implemented as a client-server architecture, typically over a network of machines. A client sends a request to the server to access a specific service. The server processes incoming requests and sends responses back to the corresponding clients. A protocol is detailed enough to allow a client to be implemented independently of a given server. In fact, a client commonly corresponds to different implementations so as to provide various subsets of the domain of services. These variations enable the client to adapt to specific constraints or user preferences. In contrast, although there usually exists different implementations of a server, these implementations tend to cover the entire domain of services.

In the IMAP case, there exists various client implementations that support the IMAP protocol, ranging from simple e-mail reading tools targeted toward

embedded systems (*e.g.*, Althea [6]) to integrated Internet environments (*e.g.*, Microsoft Outlook [7] and Netscape Messenger [8]) for workstations.

### 3.2 Variations of a Domain of Services

A given protocol may correspond to a variety of client and server implementations to account for customer needs. Although these implementations offer service variations on either a client or the server side, they must all be in compliance with the underlying protocol to make the client-server interaction possible. As such, these implementations can be viewed as a *program family*; that is, programs that share enough characteristics to be studied and developed as a whole [9]. Commonalities mainly correspond to the processing of the requests/responses that have formats and assumptions specified by the protocol. Variabilities on the server side consist of defining different semantics for client requests. On the client side, variabilities correspond to implementing different treatments for server responses.

In the IMAP case, our goal is to identify variations characterizing a scope of customized accesses to a mailbox. We explore these variations systematically by considering the various levels involved in accessing a mailbox, namely, an access-point, a mailbox, a message, and its fields (*i.e.*, message headers and message parts). At each level of this hierarchical schema, we study what programmability could be introduced with respect to the requests of the IMAP protocol. We refer to the programmability of each level as a *view*.

This hierarchical approach to views enables user preferences to be defined comprehensively: from general coarse-grained parameters, such as the terminal features, to the specific layout of a message field. A view at a given level may treat a message field as an atomic entity, *e.g.*, deciding whether to drop it. At another level, a view may trigger specific treatments for parts of a message field.

### 3.3 From Variations of a Domain of Services to a DSL

Enabling service variations to be introduced in a protocol relies on the ability to easily and safely express a variation. To this end, we propose to use DSLs as a programming paradigm. This kind of languages has been successfully developed and used in a wide spectrum of areas [10].

Many approaches, such as family analysis, have been developed to drive the design of a DSL with respect to a specific program family [11, 12, 9, 13]. These approaches aim to discover both commonalities and variations within a program family to fuel the design process of a DSL. A step toward a methodology for DSL development has been presented by Consel and Marlet [14]. Recently, this approach has been revisited to structure the development of DSLs on the notion of program family [15].

In the context of the IMAP case, we have designed a DSL, named Pems, that enables a client to define views on remote mailboxes, specifying how to adapt mailbox access to constraints or preferences like device capabilities and

available bandwidth. For example, one can filter out messages with respect to some criteria to minimize the length of the summary of new messages received.

```

view accesspoint PDA {
  Mobility = YES;
  Screen   = 320 * 240;
  Color    = NO;
  Bandwidth = 10MB/s;
  Mailbox_view = nomadic(1MB);
}

view mailbox nomadic(size s) {
  if (From == "joe@mail.fr")
    bind boss;
  if (Size > s)
    ignore;
  bind tiny;
}

view message boss {
  From as cst("The Boss!");
  Date;
  Subject;
  Body;
  Attachment[] as bwlImages(30KB);
}

view field Attachment bwlImages(size s) {
  if (Attachment.size > s)
    return "Attachment too big:" +
      Attachment.size;
  if (Attachment.type in "image/*")
    return blackwhite(Attachment.value);
  ignore;
}

```

**Fig. 1.** Pems example

As illustrated by the example shown in Figure 1, the Pems language makes it possible to define views at four different levels: access-point, mailbox, message, and message field. An access-point consists of a set of parameters such as the client terminal features, the characteristics of the link layer, and a mailbox view. A mailbox view aims to select the messages that belong to the view. It consists of a list of condition-action pairs. When a condition matches, the corresponding action is performed. An action can either drop the current message or assign it a category of messages for its processing. The message view defines a set of fields, relevant to the client, for a given category of messages. Also, a view may be assigned to some fields to trigger specific treatments. Finally, the field view aims to convert field values into some representation appropriate to the access-point.

The IMAP example illustrates how our DSL paradigm makes a protocol open to variations to cope with user needs. Thanks to the abstractions and notations provided by Pems, one can easily write customized services for accessing a mailbox. Moreover, various verifications can be performed on a service before its deployment to preserve the robustness of the underlying platform. In the IMAP case, the Pems compiler checks various program properties such as non-interference, resource usage, and confidentiality.

## 4 A Software Architecture to Support Programmability

Now that service variations can be introduced without changing the protocol, we need to study a software architecture that can support the programming of these variations. Because most protocols for communication services are implemented using a *client-server model*, we propose to examine strategies aimed to introduce programmability in this model. These adaptations of an existing software architecture should demonstrate that our DSL paradigm is a pragmatic approach to

introducing service variations. In fact, each of these strategies has been used in Nova as illustrated later in this section and in Section 5.

Although adaptations could either be done on the client side or the server side, we concentrate on the latter to relieve the client terminal from adaptation processing and the network from unnecessary data.

#### 4.1 Script-Enabled Server

Scripting languages are commonly used to introduce variations on the server side. Scripts (*e.g.*, CGI scripts [16]) can be parameterized with respect to some client data. However, this strategy is limited because scripting languages are often unrestricted, and thus, only the server administrator can introduce service variations. Such a limitation clearly contradicts our openness requirement.

One strategy to map our DSL paradigm into a script-enabled server is to compile a DSL program into a script, as proposed by the Bigwig project in the domain of Web services [17]. This strategy has two main advantages: it makes programmability more widely accessible without sacrificing robustness, and it allows the existing support for programmability to be re-used. Of course, not all servers are combined with a scripting language; and, even if they are, the purpose of this language may not coincide with the service variations that need to be addressed.

The script-enabled server approach has been used in Nova to introduce programmability in the domain of telephony services as described in Section 5.1.

#### 4.2 Proxy Server

An alternative approach to defining service variations consists of introducing a proxy server that runs client programs and invokes the unchanged server. Client programs are written in a scripting language or a general-purpose language. Robustness of the platform is guaranteed by the physical separation of the server and the proxy: they run on different machines. This approach still has a number of drawbacks. First, it consumes bandwidth for communications between the proxy and the server. Second, it requires very tight control of resource consumption (*e.g.*, CPU and memory) to prevent one script from interfering with others. Third, a buggy script can compromise the server by overflowing it with requests.

Our DSL paradigm could also be beneficial in the context of proxy servers. The idea is to have the proxy run DSL programs, as opposed to running programs written in an arbitrary language. Because the DSL programs are safe, the proxy can even run on the same machine as the one hosting the server, and thus eliminate network latency.

As described in Section 5.3, the HTTP resource adapters of Nova rely on the proxy server approach to introducing programmability.

#### 4.3 Programmable Server

To further integrate programmability in the server, we proposed to directly make the server programmable [18]. To do so, our strategy consists of enabling the se-

mantics of a request to be, to some extent, definable. The processing of a request can be seen as parameterized with respect to a service variation that takes the form of a DSL program. The DSL restrictions guarantee that service variations do not compromise server robustness. Also, in contrast with a proxy-based approach, this tight integration of service variations in the server minimizes performance overhead.

We have modified the implementation of an existing IMAP server to make it programmable. A client can introduce its own service variations in the form of a Pems program. This DSL program is deployed and run in the server, after being checked.

The implementation of Pems is traditional: it consists of a compiler and a run-time system. The compiler is a program generator that takes a Pems program and performs a number of verifications to fulfill the robustness requirements on both the server and client sides. The Pems program is then translated into C code. Finally, this C code is compiled and loaded into the server when needed.

An original IMAP server [19] has been made programmable so that code can be dynamically loaded to extend its functionalities. Yet, binding a service implementation to a particular context is a remaining key issue. Indeed, it is orthogonal to the software architecture used to support programmability. This issue is detailed elsewhere [18].

## 5 The Nova Platform

To assess our approach, we have used the DSL paradigm to develop a programmable platform for communication services, named Nova. It consists of a programmable server and a DSL for each target application domain. Five application domains are currently covered by Nova: e-mail processing, remote document processing, telephony services, streams, and HTTP resource adapters. Let us briefly present these different application areas.

### 5.1 Call Processing

Telephony services are executed over a signaling platform based on the *Session Initiation Protocol* (SIP). We have designed a dialect of C to program call processing services, named Call/C. In contrast with a prior language, called CPL [20], our DSL is a full-fledged programming language based on familiar syntax and semantics. Yet, it conforms with the features and requirements of a call processing language as listed in the RFC 2824 [21]. In fact, our DSL goes even further because it introduces domain-specific types and constructs that allow verifications beyond the reach of both CPL and general-purpose languages. The example shown in Figure 2 illustrates the use of the Call/C language to program a call forwarding service. This service is introduced by defining a behavior for the *incoming* request<sup>1</sup> of the SIP protocol. When a call is received, the incoming

---

<sup>1</sup> Strictly speaking, a call is initiated with the request *Invite* of the SIP protocol.

entry point is invoked with information about the caller. In this call forwarding service, the incoming call is redirected to `sip:dana@labri.fr`. If this redirection is itself redirected, then the location of the new call target is analyzed. If the location is not a voice-mail address, then the redirection is performed. Otherwise, the call is redirected to the callee's voice-mail `sip:john@voicemail.labri.fr`.

```
response incoming(call in) {
  response res = forward(in, sip:dana@labri.fr);
  switch(res.kind) {
    case redirect:
      if (! match(res.contact, ".*@voicemail.*") {
        return forward(in, res.contact);
      }
    default:
      return forward(in, sip:john@voicemail.labri.fr);
  }
}
```

**Fig. 2.** Call/C example

The script-enabled server approach is commonly used for programming telephony services in a SIP-based signaling platform. Examples include SIP CGI, SIP servlets and Microsoft's proprietary SIP programming API.

Being a high-level domain-specific language for telephony services, Call/C is not biased towards any particular signaling platform or telephony service programming model. This neutrality renders Call/C *retargetable* in that it may generate code for different programming layers and scripting languages.

Our current implementation of the Call/C language in Nova targets two very different programming layers, namely SIP CGI, with C as a scripting language, and SIP Servlets.

## 5.2 Remote Document Processing

Accessing a document stored on a remote server may involve various processing before getting the document in a format appropriate for a specific *sink* (i.e., a local machine or a device). The target format could depend on a variety of parameters, such as the capabilities of the access-point and the available bandwidth. To address these issues, we have developed a simple protocol for remote-document processing, named RDP, and a language aimed to define both conversion rules and sinks for documents, called RDPL.

The RDP server facilitates the process of converting documents into forms in which they are required (for example, a 1024x786 *jpeg* image with 32-bit color to a 160x200 image with 256 colors suitable for a PDA). Besides, RDP enables the server to redirect documents to specific sinks (for example, a PDF file may be redirected to a printer, or a fax machine).

An RDPL program specifies two main parts, as illustrated by the example presented in Figure 3. The first part is a list of sinks and a definition of their capabilities (e.g., overleaf printing in the case of a printer). The second part is a



filter graph that defines the intermediate steps and the filters used for the format conversion of a document. Filters can be combined to perform a wide range of transformations.

```

SINK printer1 {
  Help      "Printer on the 1st floor";
  Interface printer1_spooler;
  Capa      Overleaf    bool;
  Help      Overleaf    "Print retro/verso?";
  Capa      Priority     integer;
  Help      Priority     "0: Normal 1: High";
  Format     ps;
  Spool      /var/spool/documentqueue;
}

Formats {
  ps, txt, mp3, jpg, gif, png, ogg, pdf, wav;
}

Filters {
  txt ps enscrip;
  txt mp3 txt2mp3;
  jpg gif convert;
  jpg png convert;
  gif png convert;
  gif jpg convert;
  wav mp3 lame;
  mp3 wav mp3play_wrapper;
  wav ogg oggencode;
  ogg wav oggplay_wrapper;
}

```

Fig. 3. RDPL example

### 5.3 HTTP Resource Adaptation

More and more devices are being used to access HTTP resources (*e.g.*, PDAs, cell phones, and laptops). Such devices have different capabilities in terms of memory, computation power, graphic rendering, link layer, etc. In addition to classical HTML pages, HTTP resources may also include sound, image, and video. We have designed and developed a language, called Hades, for specifying the adaptation of HTTP resources to the features of a target access-point.

The example shown in Figure 4 illustrates the use of Hades. It specifies that video content must be removed and that images are to be replaced by a link to an image converted into gray-scale *jpeg* format.

```

accesspoint PDA {
  Mobility = YES;
  Screen   = 320 * 240;
  Color    = NO;
  Bandwidth = 10MB/s;

  image {
    changeFormat("JPEG");
    grayscale();
  }
}

html {
  image {
    externalize ("IMAGE");
  }

  video {
    remove();
  }
}

```

Fig. 4. Hades example

The ICAP protocol [22] was designed to facilitate better distribution and caching for the Web. It distributes Internet-based content from the *origin* servers, via proxy caches (ICAP clients), to dedicated ICAP servers. These ICAP servers focus on specific value-added services such as access control, authentication, language translation, content filtering, and virus scanning. Moreover, ICAP enables adaptation of content in such a way that it becomes suitable for other less powerful devices such as PDAs and mobile phones.

Our implementation of the Hades language relies on the ICAP protocol. We have developed a compiler that takes an Hades program and generates code to be loaded in an ICAP server. The Squid Web proxy is used as an ICAP client to enable the HTTP resource adaptation specified by the Hades program.

## 5.4 Stream Processing

The last application area covered by Nova is stream adaptation. We have developed a language to specify multimedia stream processing, named Spidle [23]. This language is used to program a server that adapts a stream to particular features of a target access-point.

```

filter RPE_Encoding {
  interface {
    stream inout bit[40][16] e;
    stream out bit[13][3] xMc;
    stream out bit[1][6] xmaxc;
    stream out bit[1][2] Mc;
  }
  instantiate {
    merger Padding pad;
    filter Weighting w;
    filter RPE_Grid_Selection gs;
    filter APCM_Quantization q;
    filter APCM_Inverse_Quantization iq;
    filter RPE_Grid_Positioning gp;
  }
}
[...]
```

```

[...]
```

```

map {
  e -> pad.si;
  pad.so -> w.e;
  w.x -> gs.x;
  gs.xM -> q.xM;
  gs.Mc -> gp.Mc;
  gs.Mc -> Mc;
  q.man -> iq.mant;
  q.exp -> iq.exp;
  q.xMc -> iq.xMc;
  q.xMc -> xMc;
  q.xmaxc -> xmaxc;
  iq.xMp -> gp.xMp;
  gp.ep -> e;
}
}
```

**Fig. 5.** Spidle example

The example shown in Figure 5 consists of a Spidle program that defines a network of *stream tasks*. *Flow declarations* specify how stream items flow within stream tasks (*i.e.*, graph nodes) and across stream tasks (*i.e.*, graph edges), as well as the types of these stream items.

A stream task can either be a *connector* or a *filter*. Connectors represent common patterns of value propagation. Filters correspond to transducers; they can either be *primitive* or *compound*. A primitive filter refers to an operation implemented in some other programming language. This facility enables existing filter libraries to be re-used. A compound filter is defined as a composition of stream filters and connectors. This composition is achieved by *mapping* the output stream of a task to the input stream of another task.

## 6 Assessment

In this section, we review the lessons learned from our experience in developing Nova. We provide some insights obtained from the study of the different domains of services supported by Nova. Some performance and robustness aspects are discussed and related to existing works. Finally, we address some of the issues raised by the introduction of domain-specific languages.

## 6.1 Introducing Programmability

The most favorable situation to introduce programmability is when a server is already programmable via a scripting language. A DSL can then simply be viewed as a high-level abstraction of an existing programming layer. This layer is used as the target of the DSL compiler, as shown in the case of Call/C. However, the compilation approach is only possible, if the target layer fulfills the programming requirements needed to express the desired service variations. Otherwise, a proxy-based approach can be used. This approach is interesting because, following the case of the scripting language, it does not entail changes in the origin server. This approach has been successfully used to write adaptations of HTTP resources in Hades. However, a proxy-based strategy is limited in that some functionalities require to directly manipulate the server state. Furthermore, introducing a proxy incurs either a CPU overhead, if the proxy runs on the server machine, or a network overhead, if the proxy runs on a remote machine. In the former case, an additional process is needed to adapt requests and responses, and computations may be wasted, if data are produced by the origin server but pruned by the proxy. In the latter case, communications between the server and the proxy generate network traffic that may increase latency. These potential problems motivate a third approach to introducing programmability: modifying the server to make it programmable. Beyond collapsing a server and a proxy, this approach enables programmability to reach potentially all of the functionalities of the server. Furthermore, experience shows that it does not incur significant overhead in terms of execution time, and does not introduce any network overhead, as discussed by the authors [18].

As can be observed, the DSL paradigm to programming servers is flexible and can adapt to existing infrastructures with techniques ranging from compiler retargeting to parameterization of server functionalities.

## 6.2 Performance

Traditional compilation techniques are applicable to DSLs. In fact, it has been shown that DSL features can enable drastic optimizations, beyond the reach of general-purpose languages [24].

In the IMAP case, we have conducted some experiments to assess the performance and bandwidth usage of the programmable server approach. The results of these experiments show that no significant performance overhead is introduced in the programmable IMAP server, compared to its original version [18]. In the Call/C example, we showed that DSL invariants enabled many target-specific optimizations when generating code for a given programming layer [25].

## 6.3 Robustness

Our approach assumes that service developers may not be trusted by the owner of the server. Furthermore, when the domains of services involve ordinary users, as in the case of Nova, the developer may not be an experienced programmer.

As a consequence, the DSL should guarantee specific properties so as to both preserve the integrity of the server and prevent a faulty service to corrupt or destroy user data. Notice that, most of these requirements would not be achievable in the context of general-purpose languages because of their unrestricted expressiveness [14].

In the application area of stream processing, Consel *et al.* [23] showed that the degree of robustness of a Spidle program goes beyond what can be achieved with an equivalent program written in a general-purpose language. For example, stream declarations are checked to guarantee that the composition of stream procedures are compatible with respect to both their types and the flowing direction of stream items.

## 6.4 Cost of DSL Introduction

Let us now present the key issues raised by using DSLs as a paradigm to address domains of communication services.

*Cost of DSL invention.* In our approach, a DSL is developed for each target domain of services. This systematic language invention introduces a cost in terms of domain analysis, language design and implementation. Traditionally, domain analysis and language design require significant efforts. In contrast, our approach relies on a key existing component: a protocol in the target domain. This protocol paves the way for the domain analysis by exposing the fundamental abstractions. It also suggests variations in the domain of services, feeding the language design process.

*Learning overhead.* Some effort is usually required for learning a new language. However, unlike a general-purpose language, a DSL uses domain-specific notations and constructs rather than inventing new ones. This situation increases the ability for domain experts to quickly adopt and use the language [14].

*Programming interface.* The five DSLs currently included in Nova have a textual representation and a C-like syntax. Yet, writing programs in these DSLs can use other representations. For example, one could use an existing framework such as XML to reduce the learning curve for users familiar with these notations. Also, textual forms could be abstracted by visual forms. That is, a DSL may have a graphical representation and be supported by an appropriate graphic-user interface. For example, we have developed a graphical front-end for the development of Spidle programs.

## 7 Conclusions and Future Work

Communication services are well-known to be very unpredictable and volatile. To cope with features, we propose a paradigm based on domain-specific languages. This paradigm enables networking and telecommunication experts to quickly

develop variations for a domain of services defined by a protocol. We have used this paradigm to uniformly develop a programmable platform for communication services, named Nova. Our paradigm relies on the client-server architecture to support a protocol, as is usual for communication services. We proposed various strategies to introduce programmability in this software architecture. The dedicated nature of the DSL enables critical properties to be checked on DSL programs so as to ensure the robustness of the underlying server.

Nova is currently targeted at five application areas, namely, telephony, e-mail, remote document processing, stream processing, and HTTP resource adaptation. This preliminary work suggests that our approach scales up in terms of application areas that it can cover. Moreover, the DSL approach has shown to be very effective for making properties, critical to a domain, verifiable by design of the language.

We have started studying the composability of our approach. The study has been conducted in the context of the programmable IMAP server. This server has been combined with the RDP server to transform message fields (typically attached documents) to a format that fits the capabilities of a device. These preliminary studies showed that composing programmable servers does not raise any difficulties. In fact, this is not surprising since the client-server architecture has long proved that it is composable. And, in essence, our approach just adds a setup phase to deploy a DSL program in the server. Once the deployment is done, the server behaves as usual, processing requests and sending responses.

## Acknowledgment

This work has been partly supported by the *Conseil Régional d'Aquitaine* under Contract 20030204003A.

## References

1. Ghribi, B., Logrippo, L.: Understanding GPRS: the GSM packet radio service. *Computer Networks* (Amsterdam, Netherlands: 1999) **34** (2000) 763–779
2. Mock, M., Nett, E., Schemmer, S.: Efficient reliable real-time group communication for wireless local area networks. In Hlavicka, J., Maehle, E., Pataricza, A., eds.: *Dependable Computing - EDCC-3*. Volume 1667 of *Lecture Notes in Computer Science*., Springer-Verlag (1999) 380
3. O'Mahony, D.: Umts: The fusion of fixed and mobile networking. *IEEE Internet Computing* **2** (1998) 49–56
4. IETF: Internet Message Access Protocol (IMAP) - version 4rev1 (1996) Request for Comments 2060.
5. Mullet, D., Mullet, K.: *Managing IMAP*. O'REILLY (2000)
6. Althea: An IMAP e-mail client for X Windows. <http://althea.sourceforge.net> (2002)
7. Microsoft: Microsoft Outlook. <http://www.microsoft.com/outlook> (2003)
8. Netscape: Netscape Messenger. <http://wp.netscape.com> (2003)

9. Parnas, D.: On the design and development of program families. *IEEE Transactions on Software Engineering* **2** (1976) 1–9
10. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* **35** (2000) 26–36
11. McCain, R.: Reusable software component construction: A product-oriented paradigm. In: *Proceedings of the 5th AiAA/ACM/NASA/IEEE Computers in Aerospace Conference*, Long Beach, California (1985)
12. Neighbors, J.: *Software Construction Using Components*. PhD thesis, University of California, Irvine (1980)
13. Weiss, D.: Family-oriented abstraction specification and translation: the FAST process. In: *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS)*, Gaithersburg, Maryland, IEEE Press, Piscataway, NJ (1996) 14–22
14. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. In Palamidessi, C., Glaser, H., Meinke, K., eds.: *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*. Volume 1490 of *Lecture Notes in Computer Science*, Pisa, Italy (1998) 170–194
15. Consel, C.: From a program family to a domain-specific language (2004) In this volume.
16. IETF: The WWW common gateway interface version 1.1. <http://cgi-spec.golux.com/nca> (1999) Work in progress.
17. Brabrand, C., Møller, A., Schwartzbach, M.I.: The <bigwig> project. *ACM Transactions on Internet Technology* **2** (2002)
18. Consel, C., Réveillère, L.: A programmable client-server model: Robust extensibility via dsls. In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montréal, Canada, IEEE Computer Society Press (2003) 70–79
19. University of Washington: Imap server. <ftp://ftp.cac.washington.edu/imap/> (2004)
20. Rosenberg, J., Lennox, J., Schulzrinne, H.: Programming internet telephony services. *IEEE Network Magazine* **13** (1999) 42–49
21. IETF: Call processing language framework and requirements (2000) Request for Comments 2824.
22. IETF: Internet content adaptation protocol (icap) (2003) Request for Comments 3507.
23. Consel, C., Hamdi, H., Réveillère, L., Singaravelu, L., Yu, H., Pu, C.: Spidle: A DSL approach to specifying streaming application. In: *Second International Conference on Generative Programming and Component Engineering*, Erfurt, Germany (2003)
24. Eide, E., Frei, K., Ford, B., Lepreau, J., Lindstrom, G.: Flick: A flexible, optimizing IDL compiler. In: *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, USA (1997) 44–56
25. Brabrand, C., Consel, C.: Call/c: A domain-specific language for robust internet telephony services. Research Report RR-1275-03, LaBRI, Bordeaux, France (2003)

# PiLIB: A Hosted Language for Pi-Calculus Style Concurrency

Vincent Cremet and Martin Odersky

École Polytechnique Fédérale de Lausanne  
1015 Lausanne, Switzerland  
`{vincent.cremet,martin.odersky}@epfl.ch`

**Abstract.** PiLIB is a library written in SCALA that implements the concurrency constructs of the  $\pi$ -calculus. Some features of the programming language SCALA, examined in the paper, make it possible to use almost the same syntax as in the  $\pi$ -calculus. The advantages of this library with respect to a simple  $\pi$ -calculus interpreter are that we can transmit any value along names, we can get control over them using the type system, and we have access to the full power of SCALA in terms of expressiveness and libraries.

## 1 Introduction

Support for concurrency is now almost universal in general purpose programming languages. But the supported concepts vary widely from one language to the next. For example, Concurrent ML [1] is based on an event-based model for process synchronization. Occam [2] is a programming language based on CSP [3]. In a similar way, Pict [4] is based on the  $\pi$ -calculus [5, 6]. Facile [7] also uses  $\pi$ -calculus style channel-based communication. JoCaml [8], Funnel [9], and Polyphonic C# [10] use synchronization patterns from join calculus [11]. Erlang [12] is based on an actor model where processes interpret messages in their mailboxes by pattern matching. Oz [13] uses logic variables for process communication. Id [14] and Concurrent Haskell [15] use a form of mutable variables called M-structures [16].

In contrast, most mainstream languages are based on a shared memory thread model. Recent popular languages such as Java [17] or the .NET common language runtime [18] augment this model with thread synchronization based on the Monitor concept [19, 20].

In spite of this confusing variety, some trends can be observed. A large body of theoretical research in concurrency is based on CCS or its successor, the  $\pi$ -calculus. The same holds for many specifications of concurrent systems. A  $\pi$ -calculus specification can be an executable program. Nevertheless, most concurrent systems are still implemented with shared-memory threads, which are synchronized with semaphores or monitors. Such programs are often much harder to reason about and verify than  $\pi$ -calculus specifications. But they are also often more efficient to implement.

Given this situation, it seems desirable to have a wide spectrum of programming solutions in which one can move easily from a high-level specification and prototype to a (thread-based) low-level implementation. One way to achieve this is with a programming language that has both low-level and high-level concurrency constructs. However, such a language would tend to be quite large. Moreover, in light of the great variety of high-level solutions that have been proposed it seems difficult to pick a concrete high-level syntax with confidence that it is the “right one”.

A better solution is to express the high-level language as a library abstraction in terms of the low-level one. Then the high-level language becomes, in effect, a domain-specific language where the domain is process coordination. This approach is similar to the use of skeletons in parallel programming [21, 22].

Being implemented as a library, the high-level language is embedded (or: hosted) in the low-level language. This has the advantages that no separate compiler and run-time environment needs to be designed, and that all of the facilities of the host-language can be used from the high-level language. However, if the high-level language is to be convenient to use, the host language needs to have a fair degree of expressiveness.

In this paper, we describe our experience with such an embedding. We have developed a process coordination language that is closely modeled after the  $\pi$ -calculus and that is implemented as a library in the host language Scala [23]. Scala is a new functional/object-oriented language that interacts smoothly with Java and C#. Compared to these environments, Scala has several additional language features that make it suitable as a host language for domain-specific languages. Among others, it supports the following concepts.

- A rich type system, with generics as well as abstract and dependent types.
- Object composition using mixin-style multiple inheritance.
- Named as well as anonymous functions as first-class values that can be nested.
- Pure object orientation, in the sense that every value is conceptually an object and every operator is a method call.

Scala is designed to operate in a JVM or .NET environment, so it can be regarded as an extension language for Java or C#. Scala does not have any constructs dealing with concurrency in the language proper. Instead, it re-uses the concurrency constructs of the underlying environment. These constructs are almost the same for Java and .NET – they consist in each case of a class-based thread model with monitors for synchronization.

The need to define a high-level process model for Scala came up when we taught a class on concurrency theory and concurrent programming. We wanted to be able to have students program concurrent systems that closely follow specifications in CCS and the  $\pi$ -calculus. At the same time, we wanted to relate this to Java’s classical thread/monitor model of concurrent programming. Embedding the high-level concurrent language in Scala provided an elegant means of going from specification to (classical) implementation and at the same time made available the rich facilities of the Java environment.



Implementing a high-level concurrent language as a library abstraction poses some challenges for the syntactic and type system abilities of the host language. In this paper we show how a convenient syntax and type discipline for the high-level language can be achieved with the help of Scala's constructs for abstraction and composition. Particularly important in this respect are Scala's generic type system, its ability to have arbitrary operators as methods, the dynamic, receiver-based interpretation of such operators, and its light-weight syntax for nestable anonymous functions (i.e., closures).

The rest of this paper is organized as follows. Section 2 is a short introduction to the  $\pi$ -calculus. Section 3 allows the reader to appreciate the close link between PiLIB and the  $\pi$ -calculus by comparing how a small example can be expressed in both formalisms. In Section 4 we present the elements of the PiLIB language in detail. Section 5 shows how these elements are mapped to SCALA constructs. Section 6 recapitulates the languages features necessary for the embedding. Section 7 gives a brief description of PiLIB's implementation, and Section 8 concludes.

## 2 The $\pi$ -Calculus in Short

The  $\pi$ -calculus is a model of concurrent computation in which processes can connect dynamically by exchanging the names of the channels they use to communicate. A calculus can be seen as a programming language that is limited to essential concepts in order to permit formal reasoning. So, as for programming languages, the precise definition of the  $\pi$ -calculus includes a description of the syntax of the terms that compose the language as well as a definition of their meaning.

### 2.1 Syntax

There are various versions of the  $\pi$ -calculus. The variant we chose for PiLIB has very few constructs and is similar to Milner's definition in [6]. The only differences in our definition are the absence of an unobservable action  $\tau$  and the use of recursive agent definitions instead of the replication operator  $!$ .

Here is the inductive definition of  $\pi$ -calculus processes.

#### Processes

$P, Q$	$::= \sum_{i=0}^n G_i$	Sum $(n \geq 0)$
	$\nu x. P$	Restriction
	$P \mid Q$	Parallel
	$A(y_1, \dots, y_n)$	Identifier $(n \geq 0)$

#### Guarded processes

$G$	$::= x(y).P$	Input
	$\bar{x}(y).P$	Output

#### Definitions

$$A(x_1, \dots, x_n) \stackrel{\text{def}}{=} P \quad (x_i\text{'s distinct, } \text{fn}(P) \subset \bar{x})$$

A *process* in the  $\pi$ -calculus is either a choice  $\sum G_i$  of several guarded processes  $G_i$ , or a parallel composition  $P \mid Q$  of two processes  $P$  and  $Q$ , or a restriction  $\nu x.P$  of a private fresh name  $x$  in some process  $P$ , or a process identifier  $A(y_1, \dots, y_n)$  which refers to a (potentially recursive) process definition. In a sum, processes are guarded by an input action  $x(y)$  or an output action  $\bar{x}(y)$ , which receives (respectively sends) a channel  $y$  via the channel  $x$  before the execution of the two communicating processes continues. In the process definition,  $\text{fn}(P)$  is the set of free names of the process  $P$ , i.e. the names that are not bound through the parameter of an input guarded process or through the restriction operator.

## 2.2 Structural Equivalence

A first way of assigning a meaning to processes is to define an equivalence relation between them. For instance identifying processes that differ only by the order of parallel sub-processes is a way of giving a meaning to the  $\mid$  operator. The equivalence that we define in this section is structural in the sense that it is based only on the syntax of processes. This is in contrast to behavioral equivalences which try to identify programs that have similar behaviors.

The structural equivalence is defined as the smallest congruence that contains a set of equalities. The more interesting equalities are the scope extrusion rules (EXTR-PAR, EXTR-SUM) that allow to pull a restriction to the top-level of a process, and the rule (UNFOLD) which replaces the occurrence of an agent name with the body of its definition:

$$\begin{aligned} \text{EXTR-PAR} \quad & \frac{}{P \mid \nu x.Q \equiv \nu x.(P \mid Q)} \quad x \notin \text{fn}(P) \\ \text{EXTR-SUM} \quad & \frac{}{P + \nu x.Q \equiv \nu x.(P + Q)} \quad x \notin \text{fn}(P) \\ \text{UNFOLD} \quad & \frac{A(\tilde{x}) \stackrel{\text{def}}{=} P}{A(\tilde{y}) \equiv P[\tilde{y}/\tilde{x}]} \end{aligned}$$

In the third equality,  $\tilde{x}$  represents a sequence  $x_1, \dots, x_n$  of names and  $P[\tilde{y}/\tilde{x}]$  denotes the substitution of  $y_i$  for every occurrence of  $x_i$  in  $P$ .

Beside these three equalities, there are also rules that identify processes that differ only in the names of bound variables, that declare  $+$  and  $\mid$  to be associative and commutative, and that allow permuting restriction binders  $\nu x$ .

## 2.3 Operational Semantics

The standard semantics of the  $\pi$ -calculus is given by a labeled reduction relation. Labels associated to reduction steps are needed to define bisimulation, an equivalence relation that identifies processes that can simulate each other

mutually. But to understand the computational content of a  $\pi$ -calculus term a reduction relation without labels is simpler. We therefore define  $\rightarrow$  to be the smallest relation that satisfies the following inference rules:

$$\begin{array}{c}
 \text{PAR} \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \\
 \\
 \text{NU} \frac{P \rightarrow Q}{\nu x.P \rightarrow \nu x.Q} \\
 \\
 \text{STRUCT} \frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} \\
 \\
 \text{COM} \frac{}{a(x).P + P' \mid \bar{a}(y).Q + Q' \rightarrow P[y/x] \mid Q}
 \end{array}$$

The rule PAR tells that a process can evolve independently of other processes in parallel. The rule NU is a contextual rule that makes it possible to reduce under a restriction. The rule STRUCT is the one that allows to identify equivalent processes when considering reduction. Finally, rule COM is the key to understand the synchronization mechanism of the calculus: two parallel processes that can perform complementary actions, i.e. send and receive the name of some channel along the same channel, can communicate. They proceed next with the guarded processes associated to the involved send and receive action. Note that the name  $x$  potentially free in the continuation  $P$  of the input guarded process  $a(x).P$  is bound to the transmitted name  $y$  thanks to the substitution  $[y/x]$  applied to  $P$ . The alternatives in each communicating sum that do not play a role in the communication, namely  $P'$  and  $Q'$ , are discarded, for this reason a sum is also called a *choice*.

### 3 Example: The Two-Place Buffer

#### 3.1 Specification in the $\pi$ -Calculus

The specification of a two-place buffer in the  $\pi$ -calculus consists of a set of mutually recursive process definitions:

$$\begin{aligned}
 \text{Buffer}(\text{put}, \text{get}) &= B_0(\text{put}, \text{get}) \\
 B_0(\text{put}, \text{get}) &= \text{put}(x).B_1(\text{put}, \text{get}, x) \\
 B_1(\text{put}, \text{get}, x) &= \overline{\text{get}}(x).B_0(\text{put}, \text{get}) + \text{put}(y).B_2(\text{put}, \text{get}, x, y) \\
 B_2(\text{put}, \text{get}, x, y) &= \overline{\text{get}}(x).B_1(\text{put}, \text{get}, y)
 \end{aligned}$$

The definitions  $B_0$ ,  $B_1$  and  $B_2$  represent respectively the state of the buffer when it contains zero, one or two elements, the buffer is initially empty. All the definitions are parameterized by the names of the channels used to put ( $\text{put}$ ) or

get (*get*) an element. Additionally definitions  $B_1$  and  $B_2$  are parameterized by the names of the elements stored in the buffer.

The logic of these definitions is then very intuitive: if the buffer is empty (state  $B_0$ ) it can only accept the input of an item on channel *put* and move to state  $B_1$ . If it contains exactly one item (state  $B_1$ ), it can either output this item and go back to state  $B_0$  or accept a new item and move to state  $B_2$ . Finally when the buffer contains two items it can only output the first stored item and go to state  $B_1$ .

$put(x).B_1(put, get, x)$  and  $\overline{get}\langle x \rangle.B_0(put, get)$  are two instances of *guarded processes*. The first one is an input guarded process which binds an input item on channel *put* to the name  $x$  in the continuation process  $B_1(put, get, x)$ . The second one is an output guarded process which outputs the item  $x$  on channel *get* before continuing with process  $B_0(put, get)$ .

### 3.2 Implementation with PiLIB

Because PiLIB is modeled after  $\pi$ -calculus concurrency primitives, it is possible to reproduce the specification above almost as it is:

```
def Buffer[a](put: Chan[a], get: Chan[a]): unit = {
  def B0: unit = choice ( put * { x  $\Rightarrow$  B1(x) } );
  def B1(x: a): unit = choice ( get(x) * B0, put * { y  $\Rightarrow$  B2(x, y) } );
  def B2(x: a, y: a): unit = choice ( get(x) * B1(y) );
  B0
}
```

Process definitions are replaced by function definitions (introduced by **def** in SCALA) with result type unit. This type contains just one value and corresponds to void in C or JAVA. The buffer is parameterized by the type  $a$  of the stored items.  $\text{Chan}[a]$  is the type of channels carrying objects of type  $a$ . A  $\pi$ -calculus input guarded process like  $put(x).B_1(put, get, x)$  is now written  $put * \{ x \Rightarrow B_1(x) \}$ , and an output guarded process like  $\overline{get}\langle x \rangle.B_0(put, get)$  is now written  $get(x) * B_0$  (the character  $*$  replaces the dot symbol). A  $\pi$ -calculus summation  $+$  between several alternatives is now a sequence of guarded processes separated by a comma and enclosed in choice ( ... ).

Compared to the specification in the  $\pi$ -calculus, the implementation using PiLIB has several advantages. First the channels have a type and the typechecker phase of the compiler ensures that only objects of the right type are passed to channels. Second, since processes in the  $\pi$ -calculus are modeled as functions, it is possible to hide the internal states  $B_0$ ,  $B_1$  and  $B_2$  by nesting the corresponding functions inside the function Buffer. The parameters *put* and *get* are visible from these local functions and do not need to be passed as extra parameters.

### 3.3 Using the Buffer

Here is an example of using the buffer in the  $\pi$ -calculus, where *Producer* is a process that repeatedly creates a new channel and put it in the buffer, and

*Consumer* is a process that tirelessly gets an item from the buffer and discards it.

$$\begin{aligned} \text{Producer}(\text{put}, \text{get}) &= \nu x. \overline{\text{put}}\langle x \rangle. \text{Producer}(\text{put}, \text{get}) \\ \text{Consumer}(\text{put}, \text{get}) &= \text{get}(x). \text{Consumer}(\text{put}, \text{get}) \end{aligned}$$

$$\nu \text{put}, \text{get}. \text{Producer}(\text{put}, \text{get}) \mid \text{Buffer}(\text{put}, \text{get}) \mid \text{Consumer}(\text{put}, \text{get})$$

The three processes are put in parallel using the operator  $\mid$  and are linked together through the sharing of the fresh channels *put* and *get* introduced by the restriction operator  $\nu$ .

In the example above the values added to the buffer by the producer and extracted from it by the consumer are  $\pi$ -calculus channels because there is nothing else to transmit in the  $\pi$ -calculus. So both channels and the values they carry are taken from the same domain. A typical way of typing such recursive channels in PiLIB consists in using a recursive type definition:

```
class Channel extends Chan[Channel];
```

Such a definition can be read: “A  $\pi$ -calculus channel is a channel that can carry other  $\pi$ -calculus channels”.

Using this type definition, the  $\pi$ -calculus code above has now an exact counterpart in PiLIB:

```
def Producer(put: Channel, get: Channel): unit = {
  val x = new Channel;
  choice ( put(x) * Producer(put, get) )
}
def Consumer(put: Channel, get: Channel): unit =
  choice ( get * { x  $\Rightarrow$  Consumer(put, get) } );

val put = new Channel, get = new Channel;
spawn < Producer(put, get) | Buffer(put, get) | Consumer(put, get) >
```

New features of PiLIB appear only in the way of creating new channels and executing several processes in parallel using `spawn`.

As we have seen, the implementation using PiLIB is very close to the specification in  $\pi$ -calculus. It seems as if SCALA has special syntax for  $\pi$ -calculus primitives, however PiLIB is nothing more than a library implemented using low-level concurrency constructs of SCALA inherited from JAVA. In the rest of the paper we will try to demystify the magic.

Of course we can also write a two-place buffer using monitors as in JAVA and the implementation would certainly be more efficient. But then relating the implementation to the specification would be non-trivial. With PiLIB we can closely relate the specification language and the implementation language, and thereby gain immediate confidence in our program.

## 4 Description of the PiLIB Grammar

In this section we briefly describe the primitive concurrency interface of SCALA, directly inherited from JAVA, and then introduce the PiLIB interface.

### 4.1 Original Concurrency Interface

The original concurrency interface of SCALA consists mainly of a class Monitor and a function fork with the following signatures.

```
class Monitor {
  def synchronized[a](def p: a): a;
  def wait(): unit;
  def wait(timeout: long): unit;
  def notify(): unit;
  def notifyAll(): unit;
}

def fork(def p: unit): unit;
```

The method `synchronized` in class `Monitor` executes its argument computation `p` in mutual exclusive mode – at any one time, only one thread can execute a synchronized argument of a given monitor.

A thread can suspend inside a monitor by calling the monitor’s `wait` method. At this point, the thread is suspended until a `notify` method of the same monitor is called by some other thread. If no thread is waiting on this monitor, a call to `notify` is ignored.

The `fork` method creates a new thread for the execution of its argument and returns immediately.

### 4.2 PiLIB Interface

Now we present the grammar of PiLIB. How this grammar is actually implemented as a library in SCALA is the topic of the next section. Here we just present the syntax of PiLIB constructs and give an informal description of their associated semantics.

**Channel creation.** At the basis of the PiLIB interface is the concept of *channel* (also called “name” in the  $\pi$ -calculus). A channel represents a communication medium. To get an object that represents a fresh channel that can carry objects of type  $A$ , one simply writes `new Chan[A]`.

**Guarded processes.** Assuming an expression  $a$  that evaluates to a channel carrying objects of type  $A$ , the term  $a * \{ x \Rightarrow c \}$  is an *input guarded process* with continuation  $c$ . References to  $x$  in  $c$  are bound to the value of the transmitted

object. The type of a guarded process whose continuation has result type  $B$  is  $\text{GP}[B]$ . Note that instead of  $\{x \Rightarrow c\}$  we could have used any expression of type  $A \Rightarrow B$  (the type of functions from  $A$  to  $B$ ).

Similarly, an *output guarded process* is written  $a(v) * c$  where  $v$  is the value to be sent and  $c$  is any continuation expression. The type of the guarded process is again  $\text{GP}[B]$ , where  $B$  is the type of the continuation expression  $c$ .

As in the  $\pi$ -calculus on which it is based, communications in PiLiB are synchronous: when a thread tries to output (resp. input) a value on a given channel it blocks as long as there is no other thread that tries to perform an input (resp. output) on the same channel. When two threads are finally able to communicate, through input and output guarded processes on the same channel, the thread that performs the input proceeds with the continuation of the input guarded process applied to the transmitted value, and the thread that performs the output proceeds with the continuation of the output guarded process.

**Summation.** The next specific ingredient of PiLiB is the function choice which takes an arbitrary number of guarded processes as arguments and tries to establish communication with another thread using one of these guarded processes. The choice function blocks until communication takes place, as explained above. Once the communication takes place, the guarded processes among the arguments to the function choice that do not take part in the communication are discarded.

**Forking multiple threads.** The construction `spawn` is used to fork several threads at the same time. For instance to execute concurrently some expressions  $p, q, r$ , each of type `unit`, one writes

`spawn < p | q | r >`

The figure 1 summarizes the concurrency primitives provided by PiLiB with their typing rules. All of these typing rules are admissible in the host type system.

$$\begin{array}{l}
 \text{NU} \frac{}{\text{new Chan}[A] : \text{Chan}[A]} \\
 \text{INPUT} \frac{a : \text{Chan}[A] \quad \{x \Rightarrow c\} : A \Rightarrow B}{a * \{x \Rightarrow c\} : \text{GP}[B]} \\
 \text{OUTPUT} \frac{a : \text{Chan}[A] \quad v : A \quad c : B}{a(v) * c : \text{GP}[B]} \\
 \text{CHOICE} \frac{g_i : \text{GP}[B] \quad i \in \{1, \dots, n\} \quad n \geq 0}{\text{choice } (g_1, \dots, g_n) : B} \\
 \text{SPAWN} \frac{p_i : \text{unit} \quad i \in \{1, \dots, n\} \quad n \geq 1}{\text{spawn } < p_1 \mid \dots \mid p_n > : \text{unit}}
 \end{array}$$

**Fig. 1.** PiLiB constructs

### 4.3 Derived Constructs

The class `Chan[A]` also contains methods to perform synchronous read and synchronous write. These constructs can be derived from the more primitive constructs presented so far.

If  $a$  is an expression of type `Chan[A]`, an example of a synchronous read is:

```
val x = a.read;
```

It is equivalent to (and implemented as)

```
var x: A = null;
choice (a * { y => x = y });
```

Similarly, a synchronous write `a.write(v)` is equivalent to `choice (a(v) * {})`. Each of the derived constructs corresponds to a sum containing just one alternative, what is expressed in PiLiB by a call to the function `choice` with only one argument.

## 5 Desugarization

In this section we explain how it is possible to implement PiLiB as an hosted language, i.e. as a SCALA library. We consider each PiLiB construct in turn and see how it is interpreted by the language.

### Channel Creation

In PiLiB, a fresh channel carrying objects of type  $A$  is created using the syntax `new Chan[A]`. Indeed, this is the normal SCALA syntax for creating instances of the parameterized class `Chan`.

### Input Guarded Process Expression

SCALA allows a one-argument method to be used as an infix operator, i.e. the SCALA parser will replace any expression `e f g` by `e.f(g)`. Furthermore, `*` is a valid identifier. So an input guarded process

$$a * \{ x \Rightarrow c \}$$

is recognized by the parser as

$$a.*( \{ x \Rightarrow c \} ) .$$

In the code above,  $a$  is an expression of type `Chan[A]`, and  $\{ x \Rightarrow c \}$  is an anonymous function of type  $A \Rightarrow B$ . It works because the class `Chan[T]` contains a polymorphic method named `*` with the following signature:

```
def *[U](f: T => U): GP[U];
```



The anonymous function is itself desugared as the creation of an anonymous object of type `Function1[A, B]` (if  $A$  is the type inferred for the parameter  $x$  and  $B$  the type inferred for the body  $c$ ) with a method `apply` whose unique parameter is  $x$  and whose body is  $c$ . That is,  $\{x \Rightarrow c\}$  is translated to

```
new Function1[A,B] {
  def apply(x: A): B = c;
}
```

## Output Guarded Process Expression

In SCALA an expression  $a(v)$  where  $a$  has type `Function1[T,U]` is replaced by  $a.apply(v)$ . So an expression describing an output guarded process  $a(v) * c$  is recognized as  $a.apply(v).*(c)$ . It implies that the class `Chan[A]` implements the interface `Function1[A,Product[A]]` where  $A$  is the type of the transmitted value and `Product[A]` is an helper class that contains the method `*`, which is necessary for simulating the dot symbol of the natural syntax.

## Guarded Process

The type of an input or output guarded process expression is `GP[A]` where  $A$  is the result type of the continuation.

The class `GP[A]` encapsulates the elements that compose a guarded process: the name of the channel and the continuation function for an input guarded process, and the name of the channel, the transmitted value and the continuation function for an output guarded process.

## Summation

A summation is written `choice (g1, ..., gn)` in P<sub>I</sub>LIB. It is just a call to the polymorphic function `choice` with signature:

```
def choice[A](gs: GP[A]*): A:
```

As specified by the star symbol at the end of the argument type, the argument can be a sequence of guarded processes of arbitrary length.

## Forking of Multiple Threads

The syntax to fork several threads is `spawn < p | q | r >`. As we have seen previously, this expression is recognized as `spawn.<(p).(q).(r).>`. Here is the implementation of this construct:

```
abstract class Spawn {
  def < (def p: unit): Spawn;
  def | (def p: unit): Spawn;
  def > : unit;
}
```

```

object spawn extends Spawn {
  def <(def p: unit): Spawn = { fork(p); this }
  def |(def p: unit): Spawn = { fork(p); this }
  def > : unit = ()
}

```

In the code above, `fork` is the low-level function to fork a sub-process.

The **def** modifier applied to parameters in the methods of the abstract class `Spawn` specifies that the argument must be passed without being evaluated. It will be evaluated each time it is used in the body of the function. This strategy of evaluation is called *call-by-name*. Without this indication, the different threads would be executed sequentially in the order of evaluation of the arguments of the method.

## 6 Scala Constructs Used for the Embedding

In this section we summarize the features that proved to be useful for hosting PiLIB in SCALA.

### Polymorphism

Type genericity allowed us to parameterize a channel by the type of the objects it can carry. The SCALA type system makes sure that a channel is always used with an object of the right type.

In SCALA, methods can also be polymorphic. As  $\pi$ -calculus agents are represented by methods in PiLIB, such agents can be polymorphic, like the two-place buffer in the example of Section 3.

Type parameters may have bounds in SCALA, and these bounds can be recursive. This feature known under the name of *F-bounded polymorphism* can be used to express recursive channels. For instance to define a kind of channel that can carry pairs of integers and channels of the same kind, we would write

```

class C extends Chan[Pair[int, C]]

```

### Higher-Order Functions and Anonymous Functions

We already saw that an input guarded process is expressed as a call of the method `*` with a continuation function as argument. This is an example of a higher-order function. Furthermore it is natural to express the continuation as an anonymous function because most of the time it is not recursive and does not need to be referred elsewhere in the program.

The type inference performed by the compiler allows some type annotations to be omitted, like the type of the formal parameter in the continuation of an input guarded process.

### Syntactic Sugar

It is convenient to write `x(v) * c` instead of `x.apply(v).(c)`. This is permitted by the SCALA parser which perform simple but powerful transformations.

The fact that all functions are members of a class (i.e. methods) allows us to overload already existing operators such as `*` easily without introducing ambiguity.

## Call-by-Name

The parameter modifier **def** makes it possible in SCALA to explicitly specify that some parameters must be called by name. This feature of the language is used on two occasions in PiLiB. In an output guarded process  $x(v) * c$ , the continuation  $c$  must not be evaluated at the creation of the guarded process but only after communication takes place with another matching guarded process, this is a perfect candidate to be passed by name. Also in the construct  $\text{spawn } \langle p \mid q \rangle$ , the call-by-name strategy for the parameters of the construct  $\text{spawn}$  allows to avoid a sequential evaluation of the arguments  $p$  and  $q$ .

The alternative way to delay the evaluation of an expression  $e$  consists in manipulating the closure  $() \Rightarrow e$  but it is more cumbersome for the programmer.

## Sequence Type

Another feature of SCALA, which is used in the definition of the choice function, is the possibility to pass an arbitrary number of arguments to a method.

## 7 Overview of the Implementation

To explain the implementation we first describe the data structures that are used and then present the means by which processes interact.

**Representation of guarded processes.** Two guarded processes are said *complementary* if one is an input and the other is an output on the same channel. For the sake of simplicity, the implementation unifies the concepts of input and output guarded processes: at the time where communication takes place between two complementary guarded processes, each of them provides a value to the other (the input guarded process provides the dummy value `()`) and applies its continuation to the value it receives. The data structure for a guarded process then consists of four components: a channel, a kind (input or output), the value it provides, and the continuation function to apply to the received value.

**Representation of sums.** A *sum* consists of an ordered list of guarded processes, together with the continuation to execute after communication takes place. At the creation of a sum, it is not known which of its guarded processes will react and what value this guarded process will receive, so the continuation of the sum is initially undefined. Two sums are said *complementary* if they contain complementary guarded processes. The implementation keeps a global list of pending sums, i.e. sums that are waiting to communicate. We call it the *pending list*. An invariant of this list is that the sums it contains are pairwise not complementary.

**Choice resolution.** Now we will explain what happens when a thread calls the function `choice`. The argument of the function `choice` is a sequence of guarded processes. This sequence is first turned into a sum with an undefined continuation. From now this sum is designated as the *arriving sum*. The pending list is scanned to find a complementary sum. If there is none, the arriving sum is appended to the end of the pending list, the choice function waits until the sum continuation gets a value and then execute this continuation. If there is a complementary sum, it is extracted from the pending list, both communicating sums get the value for their continuation and their respective threads are woken up.

The implementation consists of about 150 lines of SCALA code with a large part dedicated to the syntactic sugar. The complete source code is available on the web [24].

## 8 Conclusion

We have presented PiLIB, a language for concurrent programming in the style of the  $\pi$ -calculus. The language is hosted as a library in SCALA. The hosting provides “for free” a highly higher-order  $\pi$ -calculus, because all parts of a PiLIB construct are values that can be passed to methods and transmitted over channels. This is the case for channels themselves, output prefixes, continuations of input and output guarded processes, and guarded processes.

A PiLIB program corresponds closely to a  $\pi$ -calculus specification. Of course, any run of that program will only perform one of the possible traces of the specification. Moreover, this trace is not necessarily fair, because of PiLIB’s method of choosing a matching sum in the pending list. A fair implementation remains a topic for future work.

There is a result of Palamidessi [25, 26] showing that it is not possible to implement the mixed choice  $\pi$ -calculus (when a sum can contain at the same time input and output guarded processes) into an asynchronous language in a distributed (deterministic) way. Our current implementation is centralized and in this case the mixed choice is not problematic. A possible continuation of this work is to implement a distributed version of PiLIB, which would force us to abandon the mixed choice.

Our experience with PiLIB in class has been positive. Because of the close connections to the  $\pi$ -calculus, students quickly became familiar with the syntax and programming methodology. The high-level process abstractions were a big help in developing correct solutions to concurrency problems. Generally, it was far easier for the students to develop a correct system using PiLIB than using Java’s native thread and monitor-based concurrency constructs.

## References

1. Reppy, J.: CML: A higher-order concurrent language. In: Programming Language Design and Implementation, SIGPLAN, ACM (1991) 293–259
2. INMOS Ltd.: OCCAM Programming Manual. Prentice-Hall International (1984)
3. Hoare, C.A.R.: Communicating sequential processes. *Communications of the ACM* **21** (1978) 666–677 Reprinted in “Distributed Computing: Concepts and Implementations” edited by McEntire, O’Reilly and Larson, IEEE, 1984.
4. Pierce, B.C., Turner, D.N.: Pict: A programming language based on the pi-calculus. In Plotkin, G., Stirling, C., Tofte, M., eds.: *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press (2000) 455–494
5. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (Parts I and II). *Information and Computation* **100** (1992) 1–77
6. Milner, R.: *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press (1999)
7. Giacalone, A., Mishra, P., Prasad, S.: Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming* **18** (1989) 121–160
8. Conchon, S., Fessant, F.L.: Jocaml: Mobile agents for Objective-Caml. In: First International Symposium on Agent Systems and Applications (ASA’99)/Third International Symposium on Mobile Agents (MA’99), Palm Springs, CA, USA (1999)
9. Odersky, M.: Functional nets. In: *Proc. European Symposium on Programming*. Number 1782 in LNCS, Springer Verlag (2000) 1–25
10. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C<sup>#</sup>. In: *Proceedings of the 16th European Conference on Object-Oriented Programming*, Springer-Verlag (2002) 415–440
11. Fournet, C., Gonthier, G.: The reflexive chemical abstract machine and the join-calculus. In: *Principles of Programming Languages*. (1996)
12. Armstrong, J., Virding, R., Wikström, C., Williams, M.: *Concurrent Programming in Erlang*, Second Edition. Prentice-Hall (1996)
13. Smolka, G., Henz, M., Würtz, J.: Object-oriented concurrent constraint programming in Oz. In van Hentenryck, P., Saraswat, V., eds.: *Principles and Practice of Constraint Programming*. The MIT Press (1995) 29–48
14. Arvind, Gostelow, K., Plouffe, W.: The ID-Report: An Asynchronous Programming Language and Computing Machine. Technical Report 114, University of California, Irvine, California, USA (1978)
15. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In ACM, ed.: *Conference record of POPL ’96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, New York, NY, USA, ACM Press (1996) 295–308
16. Barth, P.S., Nikhil, R.S., Arvind: M-structures: Extending a parallel, non-strict, functional language with state. In Hughes, J., ed.: *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference*, Cambridge, MA, USA, Springer-Verlag (1991) 538–568 *Lecture Notes in Computer Science* 523.
17. Gosling, J., Joy, B., Steele, G., Bracha, G.: *The Java Language Specification*, Second Edition. Java Series, Sun Microsystems (2000) ISBN 0-201-31008-2.
18. Box, D.: *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley (2002)

19. Hoare, C.A.R.: Monitors: an operating system structuring concept. *Communications of the ACM* **17** (1974) 549–557
20. Hansen, P.B.: The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering* **1** (1975) 199–207
21. Bischof, al.: Generic parallel programming using c++ templates and skeletons (2004) In this volume.
22. Kuchen, H.: Optimizing sequences of skeleton calls (2004) In this volume.
23. Odersky, M.: Report on the Programming Language Scala. (2002) <http://lampwww.epfl.ch/scala/>.
24. Cremet, V.: Pilib. <http://lampwww.epfl.ch/~cremet> (2003)
25. Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In: *Proc. of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, ACM (1997) 256–265
26. Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous pi-calculus. *Mathematical Structures in Computer Science* **13** (2003) 685–719

# A Language and Tool for Generating Efficient Virtual Machine Interpreters

David Gregg<sup>1</sup> and M. Anton Ertl<sup>2</sup>

<sup>1</sup> Department of Computer Science, Trinity College, Dublin 2, Ireland  
`David.Gregg@cs.tcd.ie`

<sup>2</sup> Institut für Computersprachen, TU Wien, A-1040 Wien, Austria  
`anton@complang.tuwien.ac.at`

**Abstract.** Stack-based virtual machines (VMs) are a popular representation for implementing programming languages, and for distributing programs in a target neutral form. VMs can be implemented with an interpreter to be simple, portable, quick to start and have low memory requirements. These advantages make VM interpreters especially useful for minority-use and domain-specific languages. VM interpreters contain large amounts of repeated or similar code. Furthermore, interpreter optimisations usually involve a large number of similar changes to many parts of the interpreter source code. We present a domain-specific language for describing the instruction set architecture of a VM. Our generator takes the instruction definition and produces C code for processing the instructions in several ways: execution, VM code generation and optimisation, disassembly, tracing and profiling. The generator can apply optimisations to the generated C code for each VM instruction, and across instructions. Using profile-directed feedback and experimentation the programmer can rapidly optimise the interpreter for new architectures, environments and applications.

## 1 Introduction

Interpreters are a popular approach for implementing programming languages, especially where simplicity and ease of construction are important. A common choice when implementing interpreters is to use a virtual machine (VM), i.e., an intermediate representation similar to real machine code, which can be interpreted efficiently. Well-known examples of virtual machines are Java's JVM [1], Prolog's WAM [2], and Smalltalk's VM [3].

An unusual feature of VM interpreters when compared with other programs is that they contain large amount of repeated similar code. Writing and maintaining this code can be time consuming and expensive. Furthermore, common approaches to optimising VM interpreters often require similar (but not identical) modifications to large numbers of places in the interpreter's source code. Such optimisations must usually be performed by the programmer on the source code of the interpreter, because compilers do not have enough domain-specific knowledge of interpreters to identify such opportunities. The result is that VM interpreters

usually remain under-optimised because the programming effort and maintenance cost of experimenting with and performing these optimisations would be too high.

An alternative to writing this source code manually is to generate the repeated similar code using a program generator. In this chapter we describe ongoing work on `vmIDL` (VM Instruction Definition Language), a domain-specific language [4] for describing the instruction set architecture of a VM. Our interpreter generator `vmgen` accepts a `vmIDL` specification as input and outputs C source code for an optimised interpreter, as well as code to implement support software such as tracing, profiling, and optimising VM code at the time it is generated. The goal is to automate much of the construction of interpreters, in much the same way that routine tasks in manufacturing of physical goods are routinely automated.

This chapter presents an overview of our research on generating virtual machine generators and the relationship of our work to domain-specific program generation. Our work is not only an example of a domain-specific language and generator. It is also an enabling technology for other DSLs. By automating many routine tasks in building and optimising an interpreter, it allows portable, reasonably efficient implementations of languages to be constructed quickly.

The rest of this chapter is organised as follows. We first explain the advantages of interpreters for implementing minority-use and domain-specific languages (section 2). Section 3 makes the case for automating the construction of VM interpreters. Section 4 introduces `vmIDL`, a domain-specific language for describing virtual machine instructions. In section 5 we show how `vmgen` uses the `vmIDL` specification to generate C code for the major components of an interpreter system. Section 6 describes the three large interpreters that have been built using `vmIDL`. In section 7 optimisations performed by `vmgen` are presented. Finally, section 8 describes existing related work on interpreter generators and interpreters.

## 2 Interpreters and DSLs

Implementing a domain-specific language (DSL) is very different from implementing a widely-used general-purpose language [5]. General-purpose languages such as Fortran, C, and Java are typically rich in features, Turing complete and often have a relatively complicated syntax. Such languages usually have a large base of users, which makes it economical for sophisticated programming environments to be developed for such languages. For example, languages such as C++ have highly-optimising compilers, special syntax support in editors such Emacs, debuggers and even automatic code restructuring tools.

In contrast, domain-specific languages are smaller and simpler [6]. There are special purpose constructs to enable domain-specific optimisations or code generation. Many DSLs have a small user base. Thus, it is not economical to invest in developing sophisticated compilers and development environments. The cost of developing, maintaining and porting implementations of the language is very significant compared to the potential user base.



Interpreters have many advantages for implementing domain-specific and minority-use languages [7, 8]. First, interpreters are relatively small, simple programs. Simplicity makes them more reliable, quicker to construct and easier to maintain. Debugging an interpreter is simpler than debugging a compiler, largely because interpreters are usually much smaller programs than compilers. Second, they can be constructed to be trivially portable to new architectures. An interpreter written in a high-level language can be rapidly moved to a new architecture, reducing time to market. There are also significant advantages in different target versions of the interpreter being compiled from the same source code. The various ports are likely to be more reliable, since the same piece of source code is being run and tested on many different architectures. A single version of the source code is also significantly cheaper to maintain. Interpreters allow a fast edit-compile-run cycle, which can be very useful for explorative programming and interactive debugging. Although just in time compilers offer similar advantages, they are much more complicated, and thus expensive to construct. Finally, interpreters require much less memory than compilers, allowing them to be deployed in environments with very limited memory, a useful feature for embedded systems.

These advantages, especially simplicity and portability, have made interpreters very popular for minority-use languages, whose use is usually restricted to a limited domain. For example, Python has always been implemented using an interpreter, which has allowed a non-profit organization to implement and maintain one of the most widely ported languages available. Similarly, most implementations of Forth and Perl are based on interpreters, as are all implementations of Ruby, Logo, `sed` and `awk`. Interpreters allow a language implementation to be constructed, maintained and ported much more cheaply than using a compiler.

### 3 Automation

When creating a VM interpreter, there are many repetitive pieces of code: The code for executing one VM instruction has similarities with code for executing other VM instructions (get arguments, store results, dispatch next instruction). Similarly, when we optimise the source code for an interpreter, we apply similar transformations to the code for each VM instruction. Applying those optimisations manually would be very time consuming and expensive, and would inevitably lead us to exploring only a very small part of the design space for interpreter optimisations. This would most likely limit the performance of the resulting interpreter, because our experience is that the correct mix of interpreter optimisations is usually found by experimentation.

Our system generates C source code, which is then fed into a compiler. With respect to optimisation, there is a clear division of labour in our system. `Vmgen` performs relatively high-level optimisations while generating the source code. These are made possible by `vmgen`'s domain-specific knowledge of the structure of interpreters, and particularly of the stack. On the other hand, lower-level,

traditional tasks and optimisations such as instruction selection, register allocation and copy propagation are performed by an existing optimising C compiler. This allows us to combine the benefits of domain-specific optimisation with the advantages of product-quality compiler optimisation.

For VM code disassembly and VM code generation (i.e. generating VM code from a higher-level language, or from another format at load time), a large amount of routine, similar code also appears in interpreters. Moreover, the code for dealing with one VM instruction is distributed across several places: VM interpreter engine, VM disassembler, and VM code generation support functions. To change or add a VM instruction, typically all of these places have to be updated. These issues suggest that much of the source code for interpreters should be generated from a high-level description, rather than hand-coded using expensive programmer time.

We present `vmIDL`, a high-level domain-specific language for describing stack VM instructions. Virtual machines are often designed as stack architectures, for three main reasons: 1) It is easy to generate stack-based code from most languages; 2) stack code is very compact, requiring little space in memory; 3) stack-based code can easily be translated into other formats. Our approach combines a small DSL for describing stack effects with general purpose C code to describe how the results of the VM instruction are generated from the inputs. In addition, `vmIDL` makes it easy to write fast interpreters by supporting efficient implementation techniques and a number of optimisations.

## 4 A Domain-Specific Language

Our domain-specific language for describing VM instructions, `vmIDL`, is simple, but it allows a very large amount of routine code to be generated from a very short specification. The most important feature of `vmIDL` is that each VM instruction defines its effect on the stack. By describing the *stack effect* of each instruction at a high level, rather than as simply a sequence of low-level operations on memory locations, it is possible to perform domain-specific optimisations on accesses to the stack.

### 4.1 Instruction Specifications

A typical example of a simple instruction description is the JVM instruction `iadd`:

```
iadd ( i1 i2 -- i )
{
i = i1+i2;
}
```

The stack effect (which is described by the first line) contains the following information: the number of items popped from and pushed onto the stacks, their order, which stack they belong to (we support multiple stacks for implementing VMs such as Forth, which has separate integer and floating point stacks), their

type, and by what name they are referred to in the C code. In our example, `iadd` pops the two integers `i1` and `i2` from the data stack, executes the C code, and then pushes the integer `i` onto the data stack.

A significant amount of C code can be automatically generated from this simple stack effect description. For example, C variables are declared for each of the stack items, code to load and store items from the stack, code to write out the operands and results while tracing the execution of a program, for writing out the immediate arguments when generating VM code from source code, and when disassembling VM code. Similarly, because the effect on the stack is described at a high level, code for different low-level representations of the stack can be generated. This feature allows many of the stack optimisations described in section 7.

## 4.2 Special Keywords

In addition to the stack effects, our language also provides some keywords that can be used in the C code which have special meaning to our generator.

**SET\_IP** This keyword sets the VM instruction pointer. It is used for implementing VM branches.

**TAIL** This keyword indicates that the execution of the current VM instruction ends and the next one should be invoked. Using this keyword is only necessary when there is an early exit out of the VM instruction from within the user-supplied C code. `Vmgen` automatically appends code to invoke the next VM instruction to the end of the generated C code for each VM instruction, so **TAIL** is not needed for instructions that do not branch out early.

As an example of the use of these macros, consider a conditional branch:

```
ifeq ( #aTarget i -- )
{
    if ( i == 0 ) {
        SET_IP(aTarget);
        TAIL;
    }
}
```

The `#` prefix indicates an immediate argument. To improve branch prediction accuracy, we use **TAIL** inside the body of the `if` statement, to allow separate dispatch code for the *taken* and *not taken* (fall-through) branches of the `if` (see section 7.4).

## 4.3 Types

The type of a stack item is specified through its prefix. In our example, all stack items have the prefix `i` that indicates a 32-bit integer. The types and their prefixes are specified at the start of the `vmIDL` file:

```
\E s" int" single data-stack type-prefix i
```

---

```

vmIDL_spec → { simple_inst | super_inst | comment | declaration }
simple_inst → inst_id ( stack_effect ) { C_code }
stack_effect → {item_id} -- {item_id}
super_inst  → inst_id = inst_id {inst_id}
comment     → \ comment_string
declaration → \ E (stack_def | stack_prefix | type_prefix)
stack_def   → stack stack_id pointer_id c_type_id
stack_prefix → stack_id stack-prefix prefix_id
type_prefix → s" type_string" (single | double) stack_id type-prefix prefix_id

```

---

**Fig. 1.** Simplified EBNF grammar for vmIDL

The s" `int`" indicates the C type of the prefix (`int`). In our current implementation, this line is executable Forth code, and the slightly unorthodox s"...`int`" syntax is used to manipulate the string "`int`". The qualifier `single` indicates that this type takes only one slot on the stack, `data-stack` is the default stack for stack items of that type, and `i` is the name of the prefix. If there are several matching prefixes, the longest one is used.

#### 4.4 Programming Language Issues

Figure 1 shows a simplified grammar for vmIDL. Terminal symbols are shown in bold font. A vmIDL specification consists of zero or more instances of each of the major features of the language, which include the following. A simple instruction is a standard instruction specification of the type shown in section 4.1. It consists of the name of the instruction, a stack effect and some C code to perform the computation in the instruction.

A superinstruction is a compound instruction that consists of a sequence of simple instructions, but that incurs only the interpreter overhead of executing a single instruction (see section 7.5). A programmer specifies a superinstruction by writing a name for the superinstruction, followed by the sequence of names of the simple instructions that the superinstruction consists of. Given this simple declaration, `vmgen` automatically constructs source code to implement the superinstruction from the instruction definitions of the component instructions. No further programmer intervention is needed. Note that the grammar description of a superinstruction allows the list of simple instructions to have only one element. In a current research (i.e. unreleased) version of vmIDL, we use this feature to implement multiple versions of simple instructions (see section 7.3).

Comments in vmIDL are specified with a backslash followed by a space at the start of a line. Our work on interpreter generators originated in an implementation of the Forth language [9], and for this reason the Forth comment character is used.

The final major feature in vmIDL is a declaration. A stack declaration is used to declare the name of a stack, the name of the stack pointer used to access that stack, and the type of the data stored in that stack (typically some neutral

type, such as `void *`). When a VM uses multiple stacks, a stack prefix can be declared. Finally, type prefixes are used to identify how data on the stack should be interpreted (such as whether the value at the top of the stack should be interpreted as an integer or floating point number).

Note that the syntax for declarations is rather unusual for a programming language. As with comments, the syntax originates with Forth. The current version of our interpreter generator system is implemented in Forth, and `\E` denotes that `vmgen` should escape to the Forth interpreter. Everything appearing after the `\E` is actually executable Forth code. For example, the `vmIDL` keyword `stack` is a Forth procedure, which is called by `vmgen` to declare a new stack. Although it is intended that this escape facility will only be used for declarations, it allows our `vmgen` to be enormously flexible, since any valid Forth code can be inserted in an escape line.

A great deal of research on domain-specific languages is concerned with semantic issues such as reasoning about the properties of the described system, checking for consistency, and type systems [5]. Our work on `vmIDL` does not address these issues at all. The burden of finding semantic errors in the instruction definition falls entirely on the programmer, in much the same way as if the interpreter were written entirely in C, without the help of a generator. In fact, our current system is deliberately lightweight, with only just enough functionality to automatically generate the C code that would normally be written manually. Our experience is that this is sufficient for the purposes of building efficient VM interpreters, although occasionally we must examine the generated C code to identify errors. Our work on `vmIDL` operates under the same economics as many other domain-specific languages; the user base is not sufficiently large to support features that are not central to building the interpreter.

## 5 Generator Output

Given an instruction definition in `vmIDL`, our generator, `vmgen`, creates several different files of code, which are included into wrapper C functions using the C preprocessor `#include` feature. By generating this code from a single definition, we avoid having to maintain these different sections of code manually.

### 5.1 Interpreter Engine

Figure 2 shows the `vmgen` output for the `iadd` VM instruction. It starts with the label of the VM instruction. Note that `vmgen` supports both interpreters with switch and threaded dispatch, as well as other dispatch methods, such as using function pointers. The C macro `LABEL()` must be defined appropriately by the programmer to allow the C code to be the target of a switch, goto, or function call.

Next, the stack items used by the instruction are declared. `NEXT_P0`, `NEXT_P1`, and `NEXT_P2` are macros for the instruction dispatch sequence, which facilitate prefetching the next VM instruction (see section 7.1). The assignments following `NEXT_P0` are the stack accesses for the arguments of the VM instruction. Then the stack pointer is updated (the stacks grow towards lower addresses). Next is

---

```

LABEL(iadd) {           /* label */
int i1;                /* declarations of stack items */
int i2;
int i;
NEXT_P0;              /* dispatch next instruction (part 0) */
i1 = vm_Cell2i(sp[1]); /* fetch argument stack items */
i2 = vm_Cell2i(spTOS);
sp += 1;              /* stack pointer updates */
{                      /* user-provided C code */
i = i1+i2;
}
NEXT_P1;              /* dispatch next instruction (part 1) */
spTOS = vm_i2Cell(i); /* store result stack item(s) */
NEXT_P2;              /* dispatch next instruction (part 2) */
}

```

---

**Fig. 2.** Simplified version of the code generated for the `iadd` VM instruction

---

```

ldl    t0,8(s3) ;i1 = vm_Cell2i(sp[1]);
ldq    s2,0(s1) ;load next VM instruction
addq   s3,0x8,s3 ;sp += 1;
addq   s1,0x8,s1 ;increment VM instruction pointer
addl   t0,s4,s4 ;i = i1+i2;
jmp    (s2)     ;jump to next VM instruction

```

---

**Fig. 3.** Alpha code produced for `iadd`

the C code from the instruction specification. After that, apart from the dispatch code there is only the stack access for the result of the instruction. The stack accesses load and store values from the stack. The variable `spTOS` is used for top-of-stack caching (see section 7.2), while `vm_Cell2i` and `vm_i2Cell` are macros for changing the type of the stack item from the generic type to the type of the actual stack item. Note that if the VM instruction uses the `TAIL` keyword to exit an instruction early, then the outputted C code will contain an additional copy of the code to write results to the stack and dispatch the next instruction at the early exit point.

This C code looks long and inefficient (and the complete version is even longer, since it includes trace-collecting and other code), but GCC<sup>1</sup> optimises it quite well and produces the assembly code we would have written ourselves on most architectures we looked at, such as the Alpha code in figure 3.

---

<sup>1</sup> Other compilers (such as Intel's Compiler for Linux) usually produce similar assembly code for the stack access. Our experience is that most mainstream compilers perform copy propagation and register allocation at least as well as GCC. However, instruction dispatch is more efficient with GCC, since GNU C's labels-as-values extension can be used to implement threaded dispatch, rather than switch dispatch [10].

## 5.2 Tracing

A typical C debugger is not well suited for debugging an interpreter because the C debugger works at a too-low level and does not know anything about the interpreted program. Figure 4 shows the tracing code that we left out of figure 2. `NAME` is a macro to output the instruction name and the contents of interesting VM registers (e.g., the instruction pointer and the stack pointers). The user defines the `printarg_` functions and can thus control how the arguments and results are displayed.

---

```

LABEL(iadd) {
NAME("iadd")      /* print VM inst. name and some VM registers */
...              /* fetch stack items */
#ifdef VM_DEBUG
if (vm_debug) {
    fputs(" i1=", vm_out); printarg_i(i1); /* print arguments */
    fputs(" i2=", vm_out); printarg_i(i2);
}
#endif
...              /* user-provided C code */
#ifdef VM_DEBUG
if (vm_debug) {
    fputs(" -- ", vm_out); /* print result(s) */
    fputs(" i=", vm_out); printarg_i(i);
    fputc('\n', vm_out);
}
#endif
...              /* store stack items; dispatch */

```

---

**Fig. 4.** Tracing code generated for the `iadd` VM instruction

In addition to a tracing mechanism, we believe that a VM-level debugger would also be useful. This would allow us to set breakpoints, single-step through the program, examine the contents of the stack, instruction pointer, stack pointer, etc. A version of the interpreter that supports such interaction would be relatively easy to generate from the instruction definition, and would greatly enhance `Vmgen`'s debugging facilities. At the time of writing, the current version does not yet generate such a debugger.

## 5.3 VM Code Generation

`Vmgen` generates functions for writing VM instructions and immediate arguments to memory. For each VM instruction, a function is generated which places the opcode and any operands in memory. Using standard functions makes the code more readable and avoids error-prone repeated code to store opcodes and

operands. More importantly, using these functions allow the VM code to be automatically optimised as it is generated in memory. For example, if we generate the VM instructions `iload` followed by `iadd` and our interpreter offers the superinstruction `iload_iadd`, then these functions will automatically make the replacement. Similarly, other optimisations, such as instruction replication, that modify the VM code can also be automatically applied, at the time the VM code is generated.

## 5.4 Disassembler

Having a VM disassembler is useful for debugging the front end of the interpretive system. All the information necessary for VM disassembly is present in the instruction descriptions, so `vmgen` generates the instruction-specific parts automatically:

```
if (ip[0] == vm_inst[1]) {
    fputs("ipush", vm_out);
    fputc(' ', vm_out); printarg_i((int)ip[1]);
    ip += 2;
}
```

This example shows the code generated for disassembling the VM instruction `ipush`. The `if` condition tests whether the current instruction (`ip[0]`) is `ipush` (`vm_inst[1]`). If so, it prints the name of the instruction and its arguments, and sets `ip` to point to the next instruction. A similar piece of code is generated for all the VM's instruction set. The sequence of `ifs` results in a linear search of the existing VM instructions; we chose this approach for its simplicity and because the disassembler is not time-critical.

## 5.5 Profiling

`Vmgen` supports profiling at the VM level. The goal is to provide information to the interpreter writer about frequently-occurring (both statically and dynamically) sequences of VM instructions. The interpreter writer can then use this information to select VM instructions to replicate and sequences to combine into superinstructions.

The profiler counts the execution frequency of each basic block. At the *end of the run* the basic blocks are disassembled, and output with attached frequencies. There are scripts for aggregating this output into totals for static occurrences and dynamic execution frequencies, and to process them into superinstruction and instruction replication rules. The profiler overhead is low (around a factor of 2), allowing long-running programs to be profiled.

## 6 Experience

We have used `vmgen` to implement three interpreters: Gforth, Cacao and CVM. Our work on interpreter generators began with Forth and was later generalised



to deal with the more complicated Java VM. This section describes the three implementations, and provides a discussion of integrating a `vmIDL` interpreter into the rest of a sophisticated JVM with such features as dynamic class loading and threads.

## 6.1 The Implementations

*Gforth* [11] is a portable product-quality interpretive implementation of Forth. Forth is a stack-based language, meaning that all computations are performed by manipulating a user-visible stack. It is primarily used for low-level systems programming and embedded systems. Forth can be implemented in only a few kilobytes of memory, and the standard Forth coding style of aggressive code factoring allows extremely compact user code. Furthermore, the Forth language is designed to be parsed and compiled to VM code very efficiently, allowing interactive Forth systems in very small amounts of memory. Thus, many small embedded systems such as camera and remote sensor systems provide a small, interactive version of Forth, to allow engineers with a terminal to interact with the system easily. Perhaps the most mainstream desktop Forth application is the OpenBoot system which is used to boot all Sun Workstations.

Gforth has three programmer-visible stacks (data stack, return-stack, and floating-point stack). Most of the VM instructions are directly used as Forth words. The Gforth project started in 1992 and Gforth has been distributed as a GNU package since 1996. The current version has been ported to six different architectures, and to Unix, DOS and Windows. Gforth is the perfect example of a system where portability, simplicity, maintainability and code size are more important than execution speed. On average, Gforth is just under five times slower than BigForth [12] a popular Forth native code compiler, but is about 40% faster than Win32Forth, a widely used Forth interpreter implemented in assembly language; and more than three times faster than PFE (Portable Forth Environment), a widely-used C implementation of Forth [13].

Our second implementation is an interpreter-based variant of the *Cacao* JVM JIT compiler for the Alpha architecture [14]. The goals of building the Cacao interpreter were to see how useful `vmgen` is for implementing interpreters other than Gforth, to add any missing functionality, and to be able to measure the performance of our optimisations compared with other interpreters and JIT compilers. The Cacao interpreter performs well compared to other JVMs running on Alpha. It is more than ten times faster than the Kaffe 1.0.5 and DEC JVM 1.1.4 interpreters. On large benchmarks, the overall running time is less than 2.5 times slower than the Cacao JIT compiler. However, much of this time is spent in Cacao's slow run-time system, so the true slowdown of our interpreter over the Cacao JIT compiler is closer to a factor of 10 [13].

The Cacao interpreter implementation is rather unstable, and does not implement all aspects of the JVM standard correctly. Furthermore, as it runs only on the Alpha architecture, it is difficult to compare with other JVM implementations. For this reason, we have recently embarked on a new JVM implementation, based on Sun's CVM, a small implementation of the Java 2 Micro Edition

(J2ME) standard, which provides a core set of class libraries, and is intended for use on devices with up to 2MB of memory. It supports the full JVM instruction set, as well as full system-level threads. Our new interpreter replaces the existing interpreter in CVM. Our CVM interpreter is similar to the Cacao implementation, except that it follows the standard JVM standard fully, and it is stable and runs all benchmark programs without modification. Experimental results [15] show that on a Pentium 4 machine the Kaffe 1.0.6 interpreter is 5.76 times slower than our base version of CVM without superinstructions on standard large benchmarks. The original CVM is 31% slower, and the Hotspot interpreter, the hand-written assembly language interpreter used by Sun's Hotspot JVM is 20.4% faster than our interpreter. Finally, the Kaffe JIT compiler is just over twice as fast as our version of CVM.

## 6.2 Integration Issues

Our work originates in Forth, and a number of issues arose when implementing a full version of the JVM, which is much more complicated than Forth VMs. One important difference between Forth and the JVM is that Forth uses the same stack for all functions, whereas the JVM has a separate stack for each method. The result is that call and return instructions in the JVM must save and restore the stack pointer and stack cache. This was not particularly well supported in `vmgen` because it happens so rarely in Forth, so new features have been added to the experimental version of `vmgen` being used for the CVM implementation.

A similar problem arises with exceptions. Several JVM instructions, such as array access and integer division can throw an exception. The result is that control moves to the most recent exception handler for that type of exception, which may be in the current method, or may be in another method further up the call stack. Implementing exception handling correctly is not simple, but it is mostly orthogonal to `vmgen`. Although it appears that exceptions could complicate `vmgen`'s stack optimisations, in fact the operand stack is cleared when an exception is thrown. So while stack cache and other variables must be reloaded after an exception, it is not much more complicated than writing `vmIDL` code for method calls and returns. The complicated exception handling code must be written by the programmer outside `vmIDL`.

A more difficult problem arose with the JVM's dynamic loading and initialisation of classes. New classes can be loaded at any time, so that the currently executing method may contain references to objects of a class that has not yet been loaded. Furthermore, each class contains an initialiser which must be executed exactly the first time an object or a static field or method of that class is accessed [1]. The standard way to implement JVM instructions that can access the fields and methods of other classes is to have two versions of each such instruction. The first version loads and initialises the class, if necessary. It also finds offsets for any field or method references to avoid costly lookups on future executions. This instruction then replaces itself in the instruction stream with its corresponding *quick* version, which does not perform the initialisations, and has the necessary offsets as immediate operands, rather than symbolic references.

Our CVM implementation does not interpret original Java bytecode. Instead we take Java bytecode, and produce direct-threaded code [16] using `vmgen`'s VM code generation functions. These generated functions replace sequences of simple VM instructions with superinstructions as the VM code is generated. However, quick instructions make this process much more complicated, since the VM code modifies itself after it is created. Our current version performs another (hand-written) optimisation pass over the method each time an instruction is replaced by a quick version. This solution effective, but makes poor use of `vmgen`'s features for automatic VM code optimisation. It is not clear to us how `vmgen` can be modified to better suit Java's needs in this regard, while still remaining simple and general.

CVM uses system-level threads to implement JVM threads. Several threads can run in parallel, and in CVM these run as several different instances of the interpreter. As long as no global variables are used in the interpreter, these different instances will run independently. Implementing threads and monitors involves many difficult issues, almost all of which are made neither simpler nor more difficult by the use of `vmIDL` for the interpreter core. One exception to this was with quick instructions. The same method may be executed by simultaneously by several different threads, so race conditions can arise with quick instructions which modify the VM code. We eventually solved this problem using locks on the VM code when quickening, but the solution was not easily found. If we were to implement the system again, we would implement threading within a single instance of the interpreter, which would perform its own thread switches periodically. Interacting with the operating system's threading system is complicated, and reduces the portability of the implementation.

A final complication with our CVM interpreter arose with garbage collection. CVM implements precise garbage collection, using stack maps to identify pointers at each point where garbage collection is possible. In our implementation, at every backward branch, and at every method call, a global variable is checked to see whether some thread has requested that garbage collection should start. If it has, then the current thread puts itself into a garbage collection safe-state and waits for the collection to complete. The use of `vmIDL` neither helps nor hinders the implementation of garbage collection. Entering a safe state involves saving the stack pointer, stack cache and other variables in the same way as when a method call occurs. It seems possible that in the future, `vmgen`'s knowledge of the stack effect of each instruction could be used to help automatically generate stack maps. However, the current version contains no such feature, and items on the stack remain, essentially, untyped. The main thrust of our current `vmgen` work is interpreter optimisation, as we show in the next section.

## 7 Optimisations

This section describes a number of optimisations to improve the execution time of interpreters, and how they can be automatically applied by `vmgen` to a `vmIDL` definition.

## 7.1 Prefetching

Perhaps the most expensive part of executing a VM instruction is dispatch (fetching and executing the next VM instruction). One way to help the dispatch branch to be resolved earlier is to fetch the next instruction early. Therefore, `vmgen` generates three macro invocations for dispatch (`NEXT_P0`, `NEXT_P1`, `NEXT_P2`) and distributes them through the code for a VM instruction (see figure 2).

These macros can be defined to take advantage of specific properties of real machine architectures and microarchitectures, such as the number of registers, the latency between the VM instruction load and the dispatch jump, and autoincrement addressing mode. This scheme even allows prefetching the next-but-one VM instruction; Gforth uses this on the PowerPC architecture to good advantage (about 20% speedup).

## 7.2 Top-of-Stack Caching

`Vmgen` supports keeping the top-of-stack item (TOS) of each stack in a register (i.e., at the C level, in a local variable). This reduces the number of loads from and stores to a stack (by one each) of every VM instruction that takes one or more arguments and produces one or more results on that stack. This halves the number of data-stack memory accesses in Gforth [17]. The benefits can be seen in figure 3 (only one memory access for three stack accesses).

The downside of this optimisation is that it requires an additional register, possibly spilling a different VM register into memory. Still, we see an overall speedup of around 7%-10% for Gforth even on the register-starved IA32 (Pentium, Athlon) architecture [13]. On the PowerPC the speedup is even larger (just over 20%) as would be expected on a machine with many registers.

`Vmgen` performs this optimisation by replacing `sp[0]` with the local variable name `spTOS` when referencing stack items. Presuming the compiler allocates this variable to a register, the benefits of top-of-stack caching occur. In addition, C code is generated to flush or reload the stack for those instructions that affect the stack height without necessarily using the topmost stack element.

## 7.3 Instruction Replication

Mispredictions of indirect branches are a major component of the run-time of efficient interpreters [10]. Most current processors use a branch target buffer (BTB) to predict indirect branches, i.e., they predict that the target address of a particular indirect branch will be the same as on the last execution of the branch.

The machine code to implement a VM instruction always ends with an indirect branch to dispatch the next instruction. As long as, say, each `iload` instruction is followed by, say, an `iadd`, the indirect branch at the end of the `iload` will generally be predicted correctly. However, this is rarely the case, and it often happens that the same VM instruction appears more than once in the working set, each time with a different following VM instruction.

Instruction replication splits a VM instruction such as `iadd` into several copies: `iadd1`, `iadd2`, etc. When generating VM code and an `iadd` instruction is needed, one of the replicated versions of `iadd` is actually placed in the generated code. The different versions will have separate indirect branches, each of which is predicted separately by the BTB. Thus, the different versions can have different following VM instructions without causing mispredictions. The VM instructions to replicate are selected using profiling information. The list of instructions to replicate is included in the `vmIDL` input file, and `vmgen` automatically generates separate C source code for each replication.

We tested this optimisations on several large Forth programs, and found that it can reduce indirect branch mispredictions in Gforth by almost two thirds, and running time by around 25% [18]. The experimental version of `vmgen` that implements this optimisation uses superinstructions of length one to replicate instructions.

## 7.4 VM Branch Tails

For conditional branch VM instructions it is likely that the two possible next VM instructions are different, so it is a good idea to use different indirect branches for them. The `vmIDL` language supports this optimisation with the keyword `TAIL`. `Vmgen` expands this macro into the whole end-part of the VM instruction.

We evaluated the effect of using different indirect jumps for the different outcomes of VM conditional branches in Gforth. We found speedups of 0%–9%, with only small benefits for most programs. However, we found a small reduction in the number of executed instructions (0.6%–1.7%); looking at the assembly language code, we discovered that GCC performs some additional optimisations if we use `TAIL`.

## 7.5 Superinstructions

A superinstruction is a new, compound VM instruction that performs the work of a sequence of simple VM instructions. Superinstructions are chosen using the output of the profiler generated by `vmgen`. The list of selected sequences to make into superinstruction is included in the `vmIDL` input file by the programmer. Given this list, `vmgen` automatically generates C code to implement the superinstructions from the instruction definition of the component VM instructions.

In a superinstruction, `vmgen` keeps all intermediate stack-values in local variables (which we hope will be allocated to registers), allowing values to be passed from one component VM instruction to another without the usual loads and stores for stack accesses. In addition stack pointer updates from the different component instructions are combined, sometimes allowing the update to be eliminated entirely if the two component updates cancel one another. For example, the generated code for the superinstruction `iload-iadd` is actually shorter than that for either of its component VM instructions on x86 machines [15], because

all stack memory accesses and stack pointer updates can be eliminated. Overall, adding superinstructions gives speedups of between 20% and 80% on Gforth [13].

## 7.6 Multiple-State Stack Caching

As mentioned in section 7.2 keeping the topmost element of the stack in a register can reduce memory traffic for stack accesses by around 50%. Further gains are achievable by reserving two or more registers for stack items. The simplest way to do this is to simply keep the topmost  $n$  items in registers. For example, the local variable<sup>2</sup> `TOS_3` might store the top of stack, `TOS_2` the second from top and `TOS_1` the next item down. The problem with this approach can be seen if we consider an instruction that pushes an item onto the stack. This value of this item will be placed in the variable `TOS_3`. But first, the current value of `TOS_3` will be copied to `TOS_2`, since this is now the second from topmost item. The same applies to the value in `TOS_1`, which must be stored to memory. Thus, any operation that affects the height of the stack will result in a ripple of copies, which usually outweigh the benefits of stack caching [17].

A better solution is to introduce multiple states into the interpreter, in which the stack can be partially empty. For example, a scheme with three stack-cache registers would have four states:

- **State 0:** cache is empty
- **State 1:** one item in the cache; top-of-stack is in `TOS_1`
- **State 2:** two items in the cache; top-of-stack is in `TOS_2`
- **State 3:** three items in the cache; top-of-stack is in `TOS_3`

In this scheme, there will be four separate versions of each virtual machine instruction – one for each state. Each version of each instruction will be customised to use the correct variable names for the topmost stack items. All of this code is generated by `vmgen` automatically from the `vmIDL` definition. Figure 5 shows the output of `vmgen` for one state of the `iadd` VM instruction. The two operands are in the cache at the start of the instruction, and so are copied from `TOS_1` and `TOS_2`. The result is put into the new topmost stack location, `TOS_1`, and the state is changed<sup>3</sup> to state 1, before the dispatch of the next VM instruction. Note that there is no stack update in this instruction; the change in the height of the stack is captured by the change in state.

Multiple-state stack caching is currently implemented in an experimental, unreleased version of `vmgen`. Preliminary experiments show that memory traffic for accessing the stack can be reduced by more than three quarters using a three register cache.

<sup>2</sup> By keeping stack cache items in local variables, they become candidates to be allocated to registers. There is no guarantee that C compiler's register allocator will actually place those variables in registers, but they are likely to be good candidates because the stack cache is frequently used.

<sup>3</sup> A common way to implement the state is to use a different dispatch table or `switch` statement for each state.

---

```

LABEL(iadd_state2) {    /* label */
int i1;                /* declarations of stack items */
int i2;
int i;
NEXT_P0;               /* dispatch next instruction (part 0) */
i1 = vm_Cell2i(TOS_1); /* fetch argument stack items */
i2 = vm_Cell2i(TOS_2);
{                      /* user-provided C code */
i = i1+i2;
}
NEXT_P1;               /* dispatch next instruction (part 1) */
TOS_1 = vm_i2Cell(i);  /* store result stack item(s) */
CHANGE_STATE(1);       /* switch to state 1 */
NEXT_P2;               /* dispatch next instruction (part 2) */
}

```

---

**Fig. 5.** Simplified version of the code generated for state 2 of the `iadd` instruction with multiple-state stack caching

## 7.7 Instruction Specialisation

Many VM instructions take an immediate argument. For example, the `IGET-FIELD.QUICK` instruction loads an integer field of an object, and takes as an immediate argument the offset at which the field appears. Through profiling, we might find that a very commonly used offset is zero (indicating the first field in the object). Thus we might introduce a special version of the instruction, with the immediate operand hardwired to zero.

An experimental version of `vmgen` supports instruction specialisation. A modified version of the profiler is used to measure the values of immediate arguments on sample programs. The commonest immediate values for the most frequent instructions are selected to be specialised instructions based on the profiling information. The experimental version of `vmgen` automatically generates C source code for these specialised instructions from the instruction definition, by setting the immediate argument to a constant value, rather than loading it from the instruction stream.

Preliminary results show that specialisation has the potential to significantly improve performance, both because it reduces the work involved in executing the instruction by removing the operand fetch, and also because having several different versions of an instruction each specialized for different operands has a similar effect on indirect branch prediction as instruction replication.

## 8 Related Work

Our work on generating VM interpreters is ongoing. The best reference on the current release version of `vmgen` is [13], which gives a detailed description of `vmgen` output, and presents detailed experimental results on the performance

of the GForth and Cacao implementations. More recent work presents newer results on superinstructions and instruction replication [18] and the CVM implementation [15].

The C interpreter `hti` [19] is created using a tree parser generator and can contain superoperators. The VM instructions are specified in a tree grammar; superoperators correspond to non-trivial tree patterns. It uses a tree-based VM (linearized into a stack-based form) derived from `lcc`'s intermediate representation. A variation of this scheme is used for automatically generating interpreters of compressed bytecode [20, 21].

Many of the performance-enhancing techniques used by `vmgen` have been used and published earlier: threaded code and decoding speed [16, 22], scheduling and software pipelining the dispatch [11, 23, 24], stack caching [11, 17] and combining VM instructions [19, 25, 24, 26]. Our main contribution is to automate the implementation of these optimisations using a DSL and generator.

## 9 Conclusion

Virtual machine interpreters contain large amounts of repeated code, and optimisations require large numbers of similar changes to many parts of the source code. We have presented an overview of our work on `vmIDL`, a domain-specific language for describing the instruction sets of stack-based VMs. Given a `vmIDL` description, our interpreter generator, `vmgen`, will automatically generate the large amounts of C source code needed to implement a corresponding interpreter system complete with support for tracing, VM code generation, VM code disassembly, and profiling. Furthermore, `vmgen` will, on request, apply a variety of optimisations to the generated interpreter, such as prefetching the next VM instruction, stack caching, instruction replication, having different instances of the dispatch code for better branch prediction, and combining VM instructions into superinstructions. Generating optimised C code from a simple specification allows the programmer to experiment with optimisations and explore a much greater part of the design space for interpreter optimisations than would be feasible if the code were written manually.

### Availability

The current release version of the `vmgen` generator can be downloaded from: <http://www.complang.tuwien.ac.at/anton/vmgen/>.

## Acknowledgments

We would like to thank the anonymous reviewers for their detailed comments, which greatly improved the quality of this chapter.



## References

1. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second edn. Addison-Wesley, Reading, MA, USA (1999)
2. Ait-Kaci, H.: The WAM: A (real) tutorial. In: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991)
3. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)
4. Weiss, D.M.: Family-oriented abstraction specification and translation: the FAST process. In: Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS), Gaithersburg, Maryland, IEEE Press (1996) 14–22
5. Czarnecki, K., Eisenecker, U.: Generative programming – methods tools and applications. Addison-Wesley (2000)
6. Lengauer, C.: Program optimization in the domain of high-performance parallelism (2004) In this volume.
7. Grune, D., Bal, H., Jacobs, C., Langendoen, K.: Modern Compiler Design. Wiley (2001)
8. Ertl, M.A.: Implementation of Stack-Based Languages on Register Machines. PhD thesis, Technische Universität Wien, Austria (1996)
9. Moore, C.H., Leach, G.C.: Forth – a language for interactive computing. Technical report, Mohasco Industries, Inc., Amsterdam, NY (1970)
10. Ertl, M.A., Gregg, D.: The behaviour of efficient virtual machine interpreters on modern architectures. In: Euro-Par 2001, Springer LNCS 2150 (2001) 403–412
11. Ertl, M.A.: A portable Forth engine. In: EuroFORTH '93 conference proceedings, Mariánské Lázně (Marienbad) (1993)
12. Paysan, B.: Ein optimierender Forth-Compiler. Vierte Dimension **7** (1991) 22–25
13. Ertl, M.A., Gregg, D., Krall, A., Paysan, B.: *vmgen* – A generator of efficient virtual machine interpreters. Software – Practice and Experience **32** (2002) 265–294
14. Krall, A., Grafl, R.: CACAO – a 64 bit JavaVM just-in-time compiler. Concurrency: Practice and Experience **9** (1997) 1017–1030
15. Casey, K., Gregg, D., Ertl, M.A., Nisbet, A.: Towards superinstructions for Java interpreters. In: 7th International Workshop on Software and Compilers for Embedded Systems. LNCS 2826 (2003) 329 – 343
16. Bell, J.R.: Threaded code. Communications of the ACM **16** (1973) 370–372
17. Ertl, M.A.: Stack caching for interpreters. In: SIGPLAN '95 Conference on Programming Language Design and Implementation. (1995) 315–327
18. Ertl, M.A., Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 03), San Diego, California, ACM (2003) 278–288
19. Proebsting, T.A.: Optimizing an ANSI C interpreter with superoperators. In: Principles of Programming Languages (POPL '95). (1995) 322–332
20. Ernst, J., Evans, W., Fraser, C.W., Lucco, S., Proebsting, T.A.: Code compression. In: SIGPLAN '97 Conference on Programming Language Design and Implementation. (1997) 358–365
21. Evans, W.S., Fraser, C.W.: Bytecode compression via profiled grammar rewriting. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. (2001) 148–155
22. Klint, P.: Interpretation techniques. Software – Practice and Experience **11** (1981) 963–973

23. Hoogerbrugge, J., Augusteijn, L.: Pipelined Java virtual machine interpreters. In: Proceedings of the 9th International Conference on Compiler Construction (CC'00), Springer LNCS (2000)
24. Hoogerbrugge, J., Augusteijn, L., Trum, J., van de Wiel, R.: A code compression system based on pipelined interpreters. *Software – Practice and Experience* **29** (1999) 1005–1023
25. Piumarta, I., Riccardi, F.: Optimizing direct threaded code by selective inlining. In: SIGPLAN '98 Conference on Programming Language Design and Implementation. (1998) 291–300
26. Clausen, L., Schultz, U.P., Consel, C., Muller, G.: Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems* **22** (2000) 471–489

# Program Transformation with Stratego/XT

## Rules, Strategies, Tools, and Systems in Stratego/XT 0.9

Eelco Visser

Institute of Information and Computing Sciences, Utrecht University  
P.O. Box 80089 3508 TB, Utrecht, The Netherlands  
visser@acm.org  
<http://www.stratego-language.org>

**Abstract.** Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation systems including parsing and pretty-printing. The framework addresses the entire range of the development process; from the specification of transformations to their composition into transformation systems. This chapter gives an overview of the main ingredients involved in the composition of transformation systems with Stratego/XT, where we distinguish the abstraction levels of rules, strategies, tools, and systems.

## 1 Introduction

Program transformation, the automatic manipulation of source programs, emerged in the context of compilation for the implementation of components such as optimizers [28]. While compilers are rather specialized tools developed by few, transformation systems are becoming widespread. In the paradigm of generative programming [13], the generation of programs from specifications forms a key part of the software engineering process. In refactoring [21], transformations are used to restructure a program in order to improve its design. Other applications of program transformation include migration and reverse engineering. The common goal of these transformations is to increase programmer productivity by automating programming tasks.

With the advent of XML, transformation techniques are spreading beyond the area of programming language processing, making transformation a necessary operation in any scenario where structured data play a role. Techniques from program transformation are applicable in document processing. In turn, applications such as Active Server Pages (ASP) for the generation of web-pages in dynamic HTML has inspired the creation of program generators such as Jostraca [31], where code templates specified in the concrete syntax of the object language are instantiated with application data.

Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation

systems including parsing and pretty-printing. The framework addresses all aspects of the construction of transformation systems; from the specification of transformations to their composition into transformation systems. This chapter gives an overview of the main ingredients involved in the composition of transformation systems with Stratego/XT, where we distinguish the abstraction levels of rules, strategies, tools, and systems.

A *transformation rule* encodes a basic transformation step as a rewrite on an abstract syntax tree (Section 3). Abstract syntax trees are represented by first-order prefix terms (Section 2). To decrease the gap between the meta-program and the object program that it transforms, syntax tree fragments can be described using the concrete syntax of the object language (Section 4).

A *transformation strategy* combines a set of rules into a complete transformation by ordering their application using control and traversal combinators (Section 5). An essential element is the capability of defining traversals generically in order to avoid the overhead of spelling out traversals for specific data types. The expressive set of strategy combinators allows programmers to encode a wide range of transformation idioms (Section 6). Rewrite rules are not the actual primitive actions of program transformations. Rather these can be broken down into the more basic actions of matching, building, and variable scope (Section 7). Standard rewrite rules are context-free, which makes it difficult to propagate context information in a transformation. Scoped dynamic rewrite rules allow the run-time generation of rewrite rules encapsulating context information (Section 8).

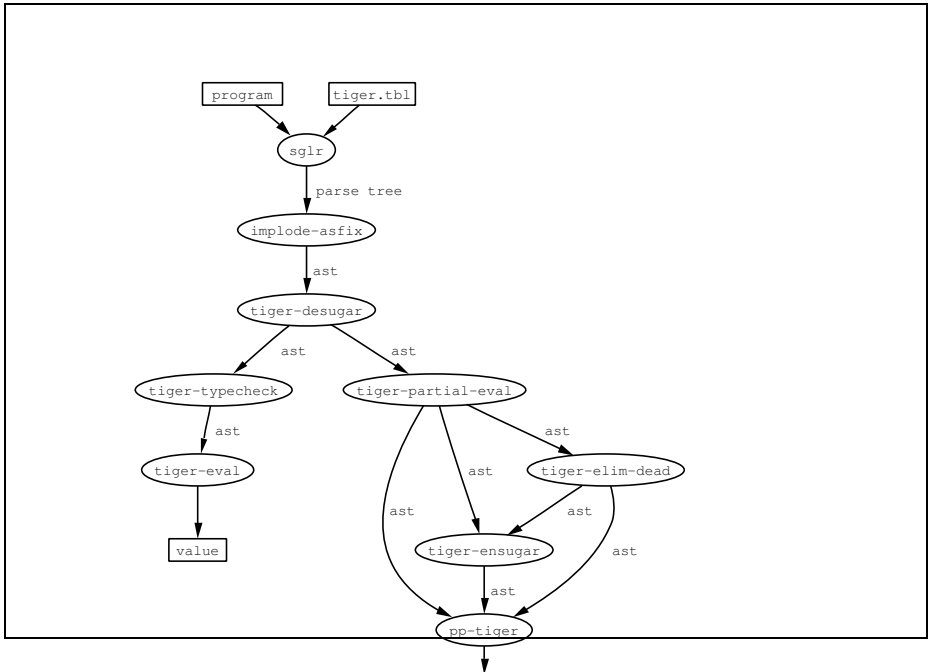
A *transformation tool* wraps a composition of rules and strategies into a stand-alone, deployable component, which can be called from the command-line or from other tools to transform terms into terms (Section 10). The use of the ATerm format makes exchange of abstract syntax trees between tools transparent.

A *transformation system* is a composition of such tools performing a complete source-to-source transformation. Such a system typically consists of a parser and a pretty-printer combined with a number of transformation tools. Figure 1 illustrates such a composition. The XTC transformation tool composition framework supports the transparent construction of such systems (Section 10).

Stratego/XT is designed such that artifacts at each of these levels of abstraction can be named and reused in several applications, making the construction of transformation systems an accumulative process. The chapter concludes with a brief overview of typical applications created using the framework (Section 12). Throughout the chapter relevant Stratego/XT publications are cited, thus providing a bibliography of the project.

## 2 Program Representation

Program transformation systems require a representation for programs that allows easy and correct manipulation. Programmers write programs as texts using text editors. Some programming environments provide more graphical (visual) interfaces for programmers to specify certain domain-specific ingredients (e.g., user interface components). But ultimately, such environments have a textual interface for specifying the details. Even if



**Fig. 1.** Composition of a transformation system from tools.

programs are written in a ‘structured format’ such as XML, the representation used by programmers generally is text. So a program transformation system needs to manipulate programs in text format.

However, for all but the most trivial transformations, a structured rather than a textual representation is needed. Bridging the gap between textual and structured representation requires parsers and unparsers. XT provides formal syntax definition with the syntax definition formalism SDF, parsing with the scannerless generalized-LR parser SGLR, representation of trees as ATerms, mapping of parse trees to abstract syntax trees, and pretty-printing using the target-independent Box language.

## 2.1 Architecture of Transformation Systems

Program transformation systems built with Stratego/XT are organized as data-flow systems as illustrated by the data-flow diagram in Figure 1 which depicts the architecture of an interpreter and a partial evaluator for Appel’s Tiger language. A program text is first parsed by *sglr*, a generic scannerless generalized-LR parser taking a parse table and a program as input, producing a parse tree. The parse tree is turned into an abstract syntax tree by *implode-asfix*. The abstract syntax tree is then transformed by one or more transformation tools. In the example, *tiger-desugar* removes and *tiger-ensugar* reconstructs syntactic sugar in a program, *tiger-typecheck* verifies the type correctness of a program and annotates its variables with type information, *tiger-eval* is an interpreter, and *tiger-partial-eval* is a partial evaluator. If the application of

these transformations results in a program, as is the case with partial evaluation, it is pretty-printed to a program text again by `pp-tiger` in the example.

## 2.2 Concrete Syntax Definition

Parsers, pretty-printers and signatures can be derived automatically from a syntax definition, a formal description of the syntax of a programming language. Stratego/XT uses the syntax definition formalism SDF [22, 34] and associated generators. An SDF definition is a declarative, integrated, and modular description of *all* aspects of the syntax of a language, including its lexical syntax. The following fragment of the syntax definition for Tiger illustrates some aspects of SDF.

```

module Tiger-Statements
imports Tiger-Lexical
exports
  lexical syntax
    [a-zA-Z][a-zA-Z0-9]*          -> Var
  context-free syntax
    Var ":" Exp                    -> Exp {cons("Assign")}
    "if" Exp "then" Exp "else" Exp -> Exp {cons("If")}
    "while" Exp "do" Exp           -> Exp {cons("While")}
    Var                             -> Exp {cons("Var")}
    Exp "+" Exp                     -> Exp {left,cons("Plus")}
    Exp "-" Exp                     -> Exp {left,cons("Minus")}
    
```

The lexical and context-free syntax of a language are described using context-free productions of the form  $s_1 \dots s_n \rightarrow s_0$  declaring that the concatenation of phrases of sort  $s_1$  to  $s_n$  forms a phrase of sort  $s_0$ . Since SDF is modular it is easy to make extensions of a language.

## 2.3 Terms and Signatures

Parse trees contain all the details of a program including literals, whitespace, and comments. This is usually not necessary for performing transformations. A parse tree is reduced to an *abstract syntax tree* by eliminating irrelevant information such as literal symbols and layout. Furthermore, instead of using sort names as node labels, *constructors* encode the production from which a node is derived. For this purpose, the productions in a syntax definition contain *constructor annotations*. For example, the abstract syntax tree corresponding to the expression  $f(a + 10) - 3$  is shown in Fig. 2. Abstract syntax trees can be represented by means of *terms*. Terms are applications  $C(t_1, \dots, t_n)$ , of a constructor  $C$  to terms  $t_i$ , lists  $[t_1, \dots, t_n]$ , strings "...", or integers  $n$ . Thus, the abstract syntax tree in the example, corresponds to the term `Minus(Call(Var("f"), [Plus(Var("a"), Int("10"))]), Int("3"))`.

The abstract syntax of a programming language or data format can be described by means of an *algebraic signature*. A signature declares for each constructor its arity  $m$ , the sorts of its arguments  $S_1 * \dots * S_m$ , and the sort of the resulting term  $S_0$  by means of a constructor declaration  $c : S_1 * \dots * S_m \rightarrow S_0$ . A term can be validated against a signature by a *format checker* [35].

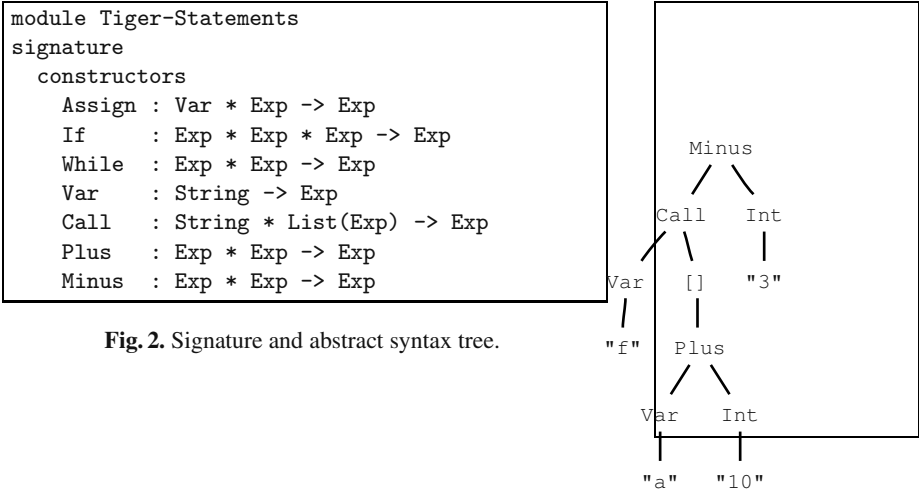


Fig. 2. Signature and abstract syntax tree.

Signatures can be derived automatically from syntax definitions. For each production  $A_1 \dots A_n \rightarrow A_0 \{ \text{cons}(c) \}$  in a syntax definition, the corresponding constructor declaration is  $c : S_1 * \dots * S_m \rightarrow S_0$ , where the  $S_i$  are the sorts corresponding to the symbols  $A_j$  after leaving out literals and layout sorts. Thus, the signature in Figure 2 describes the abstract syntax trees derived from parse trees over the syntax definition above.

2.4 Pretty-Printing

After transformation, an abstract syntax tree should be turned into text again to be useful as a program. Mapping a tree into text is the inverse of parsing, and is thus called *unparsing*. When an unparser makes an attempt at producing human readable, instead of just compiler parsable, program text, an unparser is called a *pretty-printer*. Stratego/XT uses the pretty-printing model as provided by the Generic Pretty-Printing package GPP [14]. In this model a tree is unparsed to a Box expression, which contains text with markup for pretty-printing. A Box expression can be interpreted by different back-ends to produce formatted output for different displaying devices such as plain text, HTML, and L<sup>A</sup>T<sub>E</sub>X.

3 Transformation Rules

After parsing produces the abstract syntax tree of a program, the actual transformation can be applied to it. The Stratego language is used to define transformations on terms. In Stratego, rewrite rules express basic transformations on terms.

3.1 Rewrite Rules

A rewrite rule has the form  $L : l \rightarrow r$ , where  $L$  is the label of the rule, and the term patterns  $l$  and  $r$  are its left-hand side and right-hand side, respectively. A term pattern

is either a variable, a nullary constructor  $C$ , or the application  $C(p_1, \dots, p_n)$  of an  $n$ -ary constructor  $C$  to term patterns  $p_i$ . In other words, a term pattern is a term with variables. A conditional rewrite rule is a rule of the form  $L : l \rightarrow r$  where  $s$ , with  $s$  a computation that should succeed in order for the rule to apply. An example rule is the following constant folding rule

`EvalPlus : Plus(Int(i), Int(j)) -> Int(k) where <add>(i, j) => k`

which reduces the addition of two constants to a constant by calling the library function `add` for adding integers. Another example, is the rule

`LetSplit : Let([d1, d2 | d*], e*) -> Let([d1], Let([d2 | d*], e*))`

which splits a list of let bindings into separate bindings.

### 3.2 Term Rewriting

A rule  $L: l \rightarrow r$  applies to a term  $t$  when the pattern  $l$  matches  $t$ , i.e., when the variables of  $l$  can be replaced by terms in such a way that the result is precisely  $t$ . Applying  $L$  to  $t$  has the effect of transforming  $t$  to the term obtained by replacing the variables in  $r$  with the subterms of  $t$  to which they were bound during matching. For example, applying rule `EvalPlus` to the term `Plus(Int(1), Int(2))` reduces it to `Int(3)`

Term rewriting is the exhaustive application of a set of rewrite rules to a term until no rule applies anywhere in the term. This process is also called normalization. For example, `Minus(Plus(Int(4), Plus(Int(1), Int(2))), Var("a"))` is reduced to `Minus(Int(7), Var("a"))` under repeated application of rule `EvalPlus`.

While exhaustive rewriting is the standard way that most rewriting engines apply rewrite rules, in Stratego one has to define explicitly which rules to apply in a normalization and according to which strategy. For example, a simplifier which applies a certain set of rules using the standard innermost strategy is defined as:

`simplify = innermost(EvalPlus + LetSplit + ...)`

The mechanism behind this definition will be explained in Section 5.

## 4 Concrete Object Syntax

In the previous section we saw that rewrite rules can be used to define transformations on abstract syntax trees representing the programs to be transformed, rather than on their text-based representations. But the direct manipulation of abstract syntax trees can be unwieldy for larger program fragments. Therefore, Stratego supports the specification of transformation rules using the *concrete syntax* of the object language [40]. In all places where normally a term can be written, a code fragment in the concrete syntax of the object language can be written. For example, using concrete syntax, the constant folding rule for addition can be expressed as:

`EvalPlus : |[ i + j ]| -> |[ k ]| where <add>(i, j) => k`



instead of the equivalent transformation on abstract syntax trees on the previous page. The use of concrete syntax is indicated by quotation delimiters, e.g., the `|[` and `]|` delimiters above. Note that not only the right-hand side of the rule, but also its matching left-hand side can be written using concrete syntax.

In particular for larger program fragments the use of concrete syntax makes a big difference. For example, consider the instrumentation rule

```
TraceFunction :
|[ function f(x*) : tid = e ]| ->
|[ function f(x*) : tid =
    (enterfun(f); let var x : tid in x := e; exitfun(f); x end) ]|
where new => x
```

which adds calls to `enterfun` at the entry and `exitfun` at the exit of functions. Writing this rule using abstract syntax requires a thorough knowledge of the abstract syntax and is likely to make the rule unreadable. Using concrete syntax the right-hand side can be written as a normal program fragment with holes. Thus, specification of transformation rules in the *concrete syntax* of the object language closes the conceptual distance between the programs that we write and their representation in a transformation system.

The implementation of concrete syntax for Stratego is completely generic; all aspects of the embedding of an object syntax in Stratego are user-definable including the quotation and anti-quotation delimiters and the object language itself, of course. Indeed in [40] a general schema is given for extending arbitrary languages with concrete syntax, and in [20] the application of this schema to Prolog is discussed.

## 5 Transformation Strategies

In the normal interpretation of term rewriting, terms are normalized by exhaustively applying rewrite rules to it and its subterms until no further applications are possible. But because normalizing a term with respect to *all* rules in a specification is not always desirable, and because rewrite systems need not be confluent or terminating, more careful control is often necessary. A common solution is to introduce additional constructors and use them to encode control by means of additional rules which specify where and in what order the original rules are to be applied. The underlying problem is that the rewriting strategy used by rewriting systems is fixed and implicit. In order to provide full control over the application of rewrite rules, Stratego makes the rewriting strategy *explicit* and *programmable* [27, 42, 41]. Therefore, the specification of a simplifier using innermost normalization in Section 3 required explicit indication of the rules *and* the strategy to be used in this transformation.

### 5.1 Strategy Combinators

There are many strategies that could potentially be used in program transformations, including exhaustive innermost or outermost normalization, and single pass bottom-up or topdown transformation. Instead of providing built-in implementations for each of these strategies, Stratego provides basic combinators for the composition of strategies.

Such strategies can be defined in a highly generic manner. Strategies can be parameterized with the set of rules, or in general, the transformation, to be applied by the strategy. Thus, the specification of rules can remain separate from the specification of the strategy, and rules can be reused in many different transformations.

Formally, a *strategy* is an algorithm that transforms a term into another term or fails at doing so. Strategies are composed using the following strategy combinators: sequential composition ( $s1 ; s2$ ), deterministic choice ( $s1 <+ s2$ ; first try  $s1$ , only if that fails  $s2$ ), non-deterministic choice ( $s1 + s2$ ; same as  $<+$ , but the order of trying is not defined<sup>1</sup>), guarded choice ( $s1 < s2 + s3$ ; if  $s1$  succeeds then commit to  $s2$  else  $s3$ ), testing ( $where(s)$ ; ignores the transformation achieved), negation ( $not(s)$ ; succeeds if  $s$  fails), and recursion ( $rec\ x(s)$ ).

Strategies composed using these combinators can be named using strategy definitions. A strategy definition of the form  $f(x_1, \dots, x_n) = s$  introduces a user-defined operator  $f$  with  $n$  strategy arguments, which can be called by providing it  $n$  argument strategies as  $f(s_1, \dots, s_n)$ . For example, the definition

$$try(s) = s <+ id$$

defines the combinator `try`, which applies  $s$  to the current subject term. If that fails it applies `id`, the identity strategy, to the term, which always succeeds with the original term as result. Similarly the `repeat` strategy

$$repeat(s) = try(s; repeat(s))$$

repeats transformation  $s$  until it fails. Note that strategy definitions do not explicitly mention the term to which they are applied; strategies combine term transformations, i.e., functions from terms to terms, into term transformations.

## 5.2 Term Traversal

The strategy combinators just described combine strategies which apply transformation rules at the roots of their subject terms. In order to apply a rule to a subterm of a subject term, the term must be traversed. Stratego defines several primitive operators which expose the direct subterms of a constructor application. These can be combined with the operators described above to define a wide variety of complete term traversals.

*Congruence operators* provide one mechanism for term traversal in Stratego. For each constructor  $C$  there is a corresponding congruence operator, also denoted  $C$ . If  $C$  is an  $n$ -ary constructor, then the corresponding congruence operator defines the strategy  $C(s_1, \dots, s_n)$ . Such a strategy applies only to terms of the form  $C(t_1, \dots, t_n)$ . It results in the term  $C(t'_1, \dots, t'_n)$ , provided the application of each strategy  $s_i$  to each term  $t_i$  succeeds with result  $t'_i$ . If the application of  $s_i$  to  $t_i$  fails for any  $i$ , then the application of  $C(s_1, \dots, s_n)$  to  $C(t_1, \dots, t_n)$  also fails. Congruences allow the specification of data-type specific traversals such as

$$map(s) = [] + [s \mid map(s)]$$

<sup>1</sup> Using the  $+$  operator amounts to declaring that the order in which the argument strategies are tried does not matter and that the compiler is allowed to pick any order. This is typically the case when two rules are mutually exclusive.

which applies a transformation  $s$  to the elements of a list. Another example of the use of congruences is the following control-flow strategy [29]

$$\text{control-flow}(s) = \text{Assign}(\text{id}, s) + \text{If}(s, \text{id}, \text{id}) + \text{While}(s, \text{id})$$

which applies the argument strategy  $s$ , typically a (partial) evaluation strategy, only to selected arguments in order to defer evaluation of the others.

While congruence operators support the definition of traversals that are *specific* to a data type, Stratego also provides combinators for composing *generic traversals*. The operator  $\text{all}(s)$  applies  $s$  to each of the direct subterms  $t_i$  of a constructor application  $C(t_1, \dots, t_n)$ . It succeeds if and only if the application of  $s$  to each direct subterm succeeds. In this case the resulting term is the constructor application  $C(t'_1, \dots, t'_n)$ , where each term  $t'_i$  is obtained by applying  $s$  to  $t_i$ . Note that  $\text{all}(s)$  is the identity on constants, i.e., on constructor applications without children. An example of the use of  $\text{all}$  is the definition of the strategy  $\text{bottomup}(s)$ :

$$\text{bottomup}(s) = \text{all}(\text{bottomup}(s)); s$$

The strategy expression  $(\text{all}(\text{bottomup}(s)); s)$  specifies that  $s$  is first applied recursively to all direct subterms — and thus to all subterms — of the subject term. If that succeeds, then  $s$  is applied to the resulting term. This definition of  $\text{bottomup}$  thus captures the generic notion of a bottom-up traversal over a term. Variations on this one-pass traversal are defined by the following strategies:

$$\begin{aligned} \text{topdown}(s) &= s; \text{all}(\text{topdown}(s)) \\ \text{alltd}(s) &= s <+ \text{all}(\text{alltd}(s)) \\ \text{oncetd}(s) &= s <+ \text{one}(\text{oncetd}(s)) \end{aligned}$$

$\text{Topdown}(s)$  applies  $s$  throughout a term starting at the top.  $\text{Alltd}(s)$  applies  $s$  along a frontier of a term. It tries to apply  $s$  at the root, if that succeeds the transformation is complete. Otherwise the transformation is applied recursively to all direct subterms.  $\text{Oncetd}(s)$  is similar, but uses the  $\text{one}$  combinator to apply a transformation to exactly one direct subterm. One-pass traversals such as shown above can be used in the definition of *fixpoint* traversals such as  $\text{innermost}$

$$\text{innermost}(s) = \text{bottomup}(\text{try}(s; \text{innermost}(s)))$$

which exhaustively applies a transformation  $s$  starting with innermost terms.

## 6 Transformation Idioms

The explicit control over the rewriting strategy using strategy combinators admits a wide variety of transformation idioms. In this section we discuss several such idioms to illustrate the expressiveness of strategies.

### 6.1 Cascading Transformations

The basic idiom of program transformation achieved with term rewriting is that of *cascading transformations*. Instead of applying a single complex transformation algorithm

to a program, a number of small, independent transformations are applied in combination throughout a program or program unit to achieve the desired effect. Although each individual transformation step achieves little, the cumulative effect can be significant, since each transformation feeds on the results of the ones that came before it.

One common cascading of transformations is accomplished by exhaustively applying rewrite rules to a subject term. In Stratego the definition of a cascading normalization strategy with respect to rules  $R_1, \dots, R_n$  can be formalized using an `innermost` strategy:

```
simplify =
  innermost(R1 <+ ... <+ Rn)
```

However, other strategies are possible. For example, the GHC simplifier [30] applies rules in a single traversal over a program tree in which rules are applied both on the way down and on the way up. This is expressed in Stratego by the strategy

```
simplify =
  downup(repeat(R1 <+ ... <+ Rn))
downup(s) =
  s; all(downup(s)); s
```

## 6.2 Staged Transformations

In staged computation, transformations are not applied to a subject term all at once, but rather in stages. In each stage, only rules from some particular subset of the entire set of available rules are applied. In the TAMPR program transformation system [5, 6] this idiom is called *sequence of normal forms*, since a program tree is transformed in a sequence of steps, each of which performs a normalization with respect to a specified set of rules. In Stratego this idiom can be expressed directly as

```
simplify =
  innermost(A1 <+ ... <+ Ak)
  ; innermost(B1 <+ ... <+ B1)
  ; ...
  ; innermost(C1 <+ ... <+ Cm)
```

Staged transformations can be applied fruitfully in combination with cascading transformations: a transformation is expressed as a sequence of stages, where each stage is a cascading transformation. On the other hand, the steps in a staged transformation can use quite different idioms from one another, and can even involve complex monolithic computations. The advantage of separating rules from strategies is particularly compelling in this case of staged transformations. Since rules are defined independently of the particular stages in which they are used, it is easy to reuse them in many different stages.

## 6.3 ‘Local’ Transformations

In conventional program optimization, transformations are applied throughout a program. In optimizing imperative programs, for example, complex transformations are

applied to entire programs [28]. In GHC-style compilation-by-transformation, small transformation steps are applied throughout programs. Local transformation is a style of transformation that is a mixture of these ideas. Instead of applying a complex transformation algorithm to a program we use staged, cascading transformations to accumulate small transformation steps for large effect. However, instead of applying transformations throughout the subject program, we often wish to apply them locally, i.e., only to selected parts of the subject program. This allows us to use transformations rules that would not be beneficial if applied everywhere. A typical strategy achieving such a transformation follows the pattern

```
transformation =
  alltd(
    trigger-transformation
    ; innermost(A1 <+ ... <+ An)
  )
```

The strategy `alltd(s)` descends into a term until a subterm is encountered for which the transformation `s` succeeds. In this case the strategy `trigger-transformation` recognizes a program fragment that should be transformed. Thus, cascading transformations are applied locally to terms for which the transformation is triggered. Of course more sophisticated strategies can be used for finding application locations, as well as for applying the rules locally. Nevertheless, the key observation underlying this idiom remains: Because the transformations to be applied are local, special knowledge about the subject program at the point of application can be used. This allows the application of rules that would not be otherwise applicable.

## 7 First-Class Pattern Matching

So far it was implied that the basic actions applied by strategies are rewrite rules. However, the distinction between rules and strategies is methodological rather than semantic. Rewrite rules are just syntactic sugar for strategies composed from more basic transformation actions, i.e., matching and building terms, and delimiting the scope of pattern variables [42, 41]. Making these actions first-class citizens makes many interesting idioms involving matching directly expressible.

To understand the idea, consider what happens when the following rewrite rule is applied:

```
EvalPlus : Plus(Int(i), Int(j)) -> Int(k) where <add> (i, j) => k
```

First it matches the subject term against the pattern `Plus(Int(i), Int(j))` in the left-hand side. This means that a substitution for the variables `i`, and `j` is sought, that makes the pattern equal to the subject term. If the match fails, the rule fails. If the match succeeds, the condition strategy is evaluated and the result bound to the variable `k`. This binding is then used to instantiate the right-hand side pattern `Int(k)`. The instantiated term then replaces the original subject term. Furthermore, the rule limits the scope of the variables occurring in the rule. That is, the variables `i`, `j`, and `k` are local to this rule. After the rule is applied the bindings to these variables are invisible again.

Using the primitive actions `match (?pat)`, `build (!pat)` and `scope ({ $x_1, \dots, x_n : s$ })`, this sequence of events can be expressed as

```
EvalPlus =
  {i,j,k: ?Plus(Int(i), Int(j)); where(! (i,j); add; ?k); !Int(k)}
```

The action `?pat` *matches* the current subject term against the pattern `pat`, binding all its variables. The action `!pat` *builds* the instantiation of the pattern `pat`, using the current bindings of variables in the pattern. The scope `{ $x_1, \dots, x_n : s$ }` delimits the scope of the term variables  $x_i$  to the strategy  $s$ . In fact, the Stratego compiler desugars rule definitions in this way. In general, a labeled conditional rewrite rule

```
R : p1 -> p2 where s
```

is equivalent to a strategy definition

```
R = {x1, ..., xn : ?p1; where(s); !p2}
```

with  $x_1, \dots, x_n$  the free variables of the patterns `p1` and `p2`. Similarly, the strategy application `<s> pat1 => pat2` is desugared to the sequence `!pat1; s; ?pat2`. Many other constructs such as anonymous (unlabeled) rules `\ p1 -> p2 where s \`, application of strategies in `build Int(<add>(i, j))`, contextual rules [35], and many others can be expressed using these basic actions.

## 7.1 Generic Term Deconstruction

Another generalization of pattern matching is *generic term deconstruction* [36]. Normally patterns are composed of *fixed* constructor applications  $C(p_1, \dots, p_n)$ , where the constructor name and its arity are fixed. This precludes generic transformations where the specific name of the constructor is irrelevant. Generic traversals provide a way to transform subterms without spelling out the traversal for each constructor. However, with generic traversal the structure of the term remains intact. For analysis problems, an abstract syntax tree should be turned into a value with a different structure. The term deconstruction `pat1#(pat2)` allows accessing the constructor and subterms of a term generically.

As an example, consider the strategy `exp-vars`, which collects from an expression all its variable occurrences:

```
exp-vars =
  \ Var(x) -> [x] \
  <+ \ _#(xs) -> <foldr(![], union, exp-vars)> xs \

foldr(z, c, f) =
  []; z
  <+ \ [h | t] -> <c>(<f>h, <foldr(z, c, f)>t) \
```

If the term is a variable, a singleton list containing the variable name  $x$  is produced. Otherwise the list of subterms  $xs$  is obtained using generic term deconstruction (the underscore in the pattern is a wildcard matching with any term); the variables for each subterm are collected recursively; and the union of the resulting lists is produced. Since this is a frequently occurring pattern, the `collect-om` strategy generically defines the notion of collecting outermost occurrences of subterms:

```

exp-vars =
  collect-om(?Var(_))

collect-om(s) =
  s; \ x -> [x] \
    <+ crush(![], union, collect-om(s))

crush(nul, sum, s) :
  _#(xs) -> <foldr(nul, sum, s)> xs

```

Note how `exp-vars` is redefined by passing a pattern match to `collect-om`.

## 8 Scoped Dynamic Rewrite Rules

Programmable rewriting strategies provide control over the application of rewrite rules. But a limitation of pure rewriting is that rewrite rules are context-free. That is, a rewrite rule can only use information obtained by pattern matching on the subject term or, in the case of conditional rewriting, from the subterms of the subject term. Yet, for many transformations, information from the context of a program fragment is needed. The extension of strategies with *scoped dynamic rewrite rules* [37] makes it possible to access this information.

Unlike standard rewrite rules in Stratego, dynamic rules are generated at run-time, and can access information available from their generation contexts. For example, in the following strategy, the transformation rule `InlineFun` defines the replacement of a function call `f(a*)` by the appropriate instantiation of the body `e1` of its definition:

```

DeclareFun =
  ?fdec@|[ function f(x1*) ta = e1 ]|;
  rules(
    InlineFun :
      |[ f(a*) ]| -> |[ let d* in e2 end ]|
      where <rename>fdec => |[ function f(x2*) ta = e2 ]|
        ; <zip(BindVar)>(x2*, a*) => d*
  )
  BindVar :
    (FArg |[ x ta ]|, e) -> |[ var x ta := e ]|

```

The rule `InlineFun` is generated by `DeclareFun` in the context of the *definition* of the function `f`, but applied at the *call sites* `f(a*)`. This is achieved by declaring `InlineFun` in the scope of the match to the function definition `fdec` (second line); the variables bound in that match, i.e., `fdec` and `f`, are inherited by the `InlineFun` rule declared within the `rules(...)` construct. Thus, the use of `f` in the left-hand side of the rule and `fdec` in the condition refer to inherited bindings to these variables.

Dynamic rules are first-class entities and can be applied as part of a global term traversal. It is possible to restrict the application of dynamic rules to certain parts of subject terms using rule scopes, which limit the live range of rules. For example, `DeclareFun` and `InlineFun` as defined above, could be used in the following simple inlining strategy:

```

inline = {| InlineFun
        : try(DeclareFun)
        ; repeat(InlineFun + Simplify)
        ; all(inline)
        ; repeat(Simplify)
        |}

```

This transformation performs a single traversal over an abstract syntax tree. First inlining functions are generated for all functions encountered by `DeclareFun`, function calls are inlined using `InlineFun`, and expressions are simplified using some set of `Simplify` rules. Then the tree is traversed using `all` with a recursive call of the inliner. Finally, on the way up, the simplification rules are applied again. The dynamic rule scope  $\{ | L : s | \}$  restricts the scope of a generated rule  $L$  to the strategy  $s$ . Of course an actual inliner will be more sophisticated than the strategy shown above; most importantly an inlining criterium should be added to `DeclareFun` and/or `InlineFun` to determine whether a function should be inlined at all. However, the main idea will be the same.

After generic traversal, dynamic rules constituted a key innovation of Stratego that allow many more transformation problems to be addressed with the idiom of strategic rewriting. Other applications of dynamic rules include bound variable renaming [37], dead-code elimination [37], constant-propagation [29] and other data-flow optimizations, instruction selection [9], type checking, partial evaluation, and interpretation [19].

## 9 Term Annotations

Stratego uses terms to represent the abstract syntax of programs or documents. A term consists of a constructor and a list of argument terms. Sometimes it is useful to record additional information about a term without adapting its structure, i.e., creating a constructor with additional arguments. For this purpose terms can be annotated. Thus, the results of a program analysis can be stored directly in the nodes of the tree.

In Stratego a term always has a list of annotations. This is the empty list if a term does not have any annotations. A term with annotations has the form  $t\{a_1, \dots, a_m\}$ , where  $t$  is a term as defined in Section 2, the  $a_i$  are terms used as annotations, and  $m \geq 0$ . A term  $t\{\}$  with an empty list of annotations is equivalent to  $t$ . Since annotations are terms, any transformations defined by rules and strategies can be applied to them.

The annotations of a term can be retrieved in a pattern match and attached in a build. For example the build `!Plus(1, 2){Int}` will create a term `Plus(1, 2)` with the term `Int` as the only annotation. Naturally, the annotation syntax can also be used in a match: `?Plus(1, 2){Int}`. Note however that this match only accepts `Plus(1, 2)` terms with just one annotation, which should be the empty constructor application `Int`. This match will thus not allow other annotations. Because a rewrite rule is just sugar for a strategy definition, the usage of annotations in rules is just as expected. For example, the rule

```

TypeCheck : Plus(e1{Int}, e2{Int}) -> Plus(e1, e2){Int}

```

checks that the two subterms of the `Plus` have annotation `Int` and then attaches the annotation `Int` to the whole term. Such a rule is typically part of a typechecker which



checks type correctness of the expressions in a program *and* annotates them with their types. Similarly many other program analyses can be expressed as program transformation problems. Actual examples in which annotations were used include escaping variables analysis in a compiler for an imperative language, strictness analysis for lazy functional programs, and bound-unbound variables analysis for Stratego itself.

Annotations are useful to store information in trees without changing their signature. Since this information is part of the tree structure it is easily made persistent for exchange with other transformation tools (Section 10). However, annotations also bring their own problems. First of all, transformations are expected to preserve annotations produced by different transformations. This requires that traversals preserve annotations, which is the case for Stratego's traversal operators. However, when transforming a term it is difficult to preserve the annotations on the original term since this should be done according to the semantics of the annotations. Secondly, it is no longer straightforward to determine the equality relation between terms. Equality can be computed with or without (certain) annotations. These issues are inherent in any annotation framework and preclude smooth integration of annotations with the other features discussed; further research is needed in this area.

## 10 Transformation Tools and Systems

A transformation defined using rewrite rules and strategies needs to be applied to actual programs. That is, it needs to read an input program, transform it, and write an output program. In addition, it needs to take care of command-line options such as the level of optimization. The Stratego Standard Library provides facilities for turning a transformation on terms into a transformation on files containing programs or intermediate representations of programs.

*ATerm Exchange Format.* The terms Stratego uses internally correspond exactly with terms in the ATerm exchange format [7]. The Stratego run-time system is based on the ATerm Library which provides support for internal term representation as well as their persistent representation in files, making it trivial to provide input and output for terms in Stratego, and to exchange terms between transformation tools. Thus, transformation systems can be divided into small, reusable tools

*Foreign Function Interface.* Stratego has a foreign function interface which makes it possible to call C functions from Stratego functions. The operator `prim( $f, t_1, \dots, t_n$ )` calls the C function  $f$  with term arguments  $t_i$ . Via this mechanism functionality such as arithmetic, string manipulation, hash tables, I/O, and process control are incorporated in the library without having to include them as built-ins in the language. For example, the definition

```
read-from-stream =
  ?Stream(stream)
  ; prim("SSL_read_term_from_stream", stream)
```

introduces an alias for a primitive reading a term from an input stream. In fact several language features started their live as a collection of primitives before being elevated to the level of language construct; examples are dynamic rules and annotations.

*Wrapping Transformations in Tools.* To make a transformation into a tool, the Stratego Standard Library provides abstractions that take care of all I/O issues. The following example illustrates how a `simplify` strategy is turned into a tool:

```
module simplify
imports lib Tiger-Simplify
strategies
  main = io-wrap(simplify-options, simplify)
  simplify-options =
    ArgOption("-O", where(<set-config> ("-O", <id>)),
              !"-O n      Set optimization level (1 default)")
```

The main strategy represents the tool. It is defined using the `io-wrap` strategy, which takes as arguments the non-default command-line options and the strategy to apply. The wrapper strategy parses the command-line options, providing a standardized tool interface with options such as `-i` for the input and `-o` for the output file. Furthermore, it reads the input term, applies the transformation to it, and writes the resulting term to output. Thus, all I/O complexities are hidden from the programmer.

*Tool Collections.* Stratego's usage of the ATerm exchange format and its support for interface implementation makes it very easy to make small reusable tools. In the spirit of the Unix pipes and filters model, these tools can be mixed and matched in many different transformation systems. However, instead of transforming text files, these tools transform structured data. This approach has enabled and encouraged the construction of a large library of reusable tools. The core library is the XT bundle of transformation tools [17], which provides some 100 more or less generic tools useful in the construction and generation of transformation systems. This includes the implementation of pretty-printing formatters of the generic pretty-printing package GPP [14], coupling of Stratego transformation components with SDF parsers, tools for parsing and pretty-printing, and generators for deriving components of transformation systems from a syntax definition. A collection of application-specific transformation components based on the XT library is emerging (see Section 12).

*Transformation Tool Composition.* A transformation system implements a complete source-to-source transformation, while tools just implement an aspect. Construction of complete transformation systems requires the composition of transformation tools. For a long time composition of transformation tools in XT was done using conventional means such as makefiles and shell scripts. However, these turn out to cause problems with determining the installation location of a tool requiring extensive configuration, transforming terms at the level of the composition, and poor abstraction and control facilities.

The XTC model for transformation tool composition was designed in order to alleviate these problems. Central in the XTC model is a repository which registers the locations of specific versions of tools. This allows a much more fine-grained search than is possible with directory-based searches. A library of abstractions implemented in Stratego supports transparently calling tools. Using the library a tool can be applied just like a basic transformation step. All the control facilities of Stratego can be used

```

io-tiger-pe =
  xtc-io-wrap(tiger-pe-options,
    parse-tiger
  ; tiger-desugar
  ; tiger-partial-eval
  ; if-switch(!"elim-dead", tiger-elim-dead)
  ; if-switch(!"ensugar",   tiger-ensugar)
  ; if-switch(!"pp",       pp-tiger)
  )
tiger-partial-eval =
  xtc-transform(!"Tiger-Partial-Eval", pass-verbose)
...

```

**Fig. 3.** Example transformation tool composition.

in their composition. Figure 3 illustrates the use of XTC in the composition of a partial evaluator from transformation components, corresponding to the right branch of the data-flow diagram in Figure 1.

## 11 Stratego/XT in Practice

The Stratego language is implemented by means of a compiler that translates Stratego programs to C programs. Generated programs depend on the ATerm library and a small Stratego-specific, run-time system. The Stratego Standard Library provides a large number of generic and data-type specific reusable rules and strategies. The compiler and the library, as well as number of other packages from the XT collection are bundled in the Stratego/XT distribution, which is available from [www.stratego-language.org](http://www.stratego-language.org) [43] under the LGPL license. The website also provides user documentation, pointers to publications and applications, and mailinglists for users and developers.

## 12 Applications

The original application area of Stratego is the specification of optimizers, in particular for functional compilers [42]. Since then, Stratego has been applied in many areas of language processing:

- Compilers: typechecking, translation, desugaring, instruction selection
- Optimization: data-flow optimizations, vectorization, ghc-style simplification, deforestation, domain-specific optimization, partial evaluation, specialization of dynamic typing
- Program generators: pretty-printer and signature generation from syntax definitions, application generation from DSLs, language extension preprocessors
- Program migration: grammar conversion
- Program understanding: documentation generation
- Document generation and transformation: XML processing, web services

The rest of this section gives an overview of applications categorized by the type of the source language.

*Functional Languages.* Simplification in the style of the Glasgow Haskell Compiler [30] was the first target application for Stratego [42], and has been further explored for the language Mondrian and recently in an optimizer for the Helium compiler. Other optimizations for functional programs include an implementation of the warm fusion algorithm for deforestation of lazy functional programs [23], and a partial evaluator for a subset of Scheme (similix) [32].

*Imperative Languages.* Tiger is the example language of Andrew Appel's text book on compiler construction. It has proven a fruitful basis for experimentation with all kinds of transformations and for use in teaching [43]. Results include techniques for building interpreters [19], implementing instruction selection (maximal munch and burg-style dynamic programming) [9], and specifying optimizations such as function inlining [37] and constant propagation [29].

These techniques are being applied to real imperative languages. CodeBoost [2] is a transformation framework for the domain-specific optimization of C++ programs developed for the optimization of programs written in the Sophus style. Several application generators have been developed for the generation of Java and C++ programs.

*Transformation Tools.* The Stratego compiler is bootstrapped, i.e., implemented in Stratego, and includes desugaring, implementation of concrete syntax, semantic checks, pattern match compilation, translation to C, and various optimizations [24].

Stratego is used as the implementation language for numerous meta-tools in the XT bundle of program transformation tools [17]. This includes the implementation of pretty-printing formatters of the generic pretty-printing package GPP [14] and the coupling of Stratego transformation components with SDF parsers.

*Other Languages.* In a documentation generator for SDL [16], Stratego was used to extract transition tables from SDL specifications.

*XML and Meta-data for Software Deployment.* The structured representation of data, their easy manipulation and external representation, makes Stratego an attractive language for processing XML documents and other structured data formats. For example, the Autobundle system [15] computes compositions (bundles) of source packages by analyzing the dependencies in package descriptions represented as terms and generates an online package base from such descriptions. Application in other areas of software deployment is underway. The generation of XHTML and other XML documents is also well supported with concrete syntax for XML in Stratego and used for example in xDoc, a documentation generator for Stratego and other languages.

## 13 Related Work

Term rewriting [33] is a long established branch of theoretical computer science. Several systems for program transformation are based on term rewriting. The motivation for and the design of Stratego were directly influenced by the ASF+SDF and ELAN languages. The algebraic specification formalism ASF+SDF [18] is based on pure rewriting with concrete syntax without strategies. Recently traversal functions were added to

ASF+SDF to reduce the overhead of traversal control [8]. The ELAN system [4] first introduced the ideas of user-definable rewriting strategies in term rewriting systems. However, generic traversal is not provided in ELAN. The first ideas about programmable rewriting strategies with generic term traversal were developed with ASF+SDF [27]. These ideas were further developed in the design of Stratego [42, 41]. Also the generalization of concrete syntax [40], first-class pattern matching [35], generic term deconstruction [36], scoped dynamic rewrite rules [37], annotations, and the XTC component composition model are contributions of Stratego/XT. An earlier paper [38] gives a short overview of version 0.5 of the Stratego language and system, before the addition of concrete syntax, dynamic rules, and XTC.

Other systems based on term rewriting include TAMPR [5, 6] and Maude [11, 10]. There are also a large number of transformation systems not based (directly) on term rewriting, including TXL [12] and JTS [3]. A more thorough discussion of the commonalities between Stratego and other transformation systems is beyond the scope of this paper. The papers about individual language concepts cited throughout this paper discuss related mechanisms in other languages. In addition, several papers survey aspects of strategies and related mechanisms in programming languages. A survey of strategies in program transformation systems is presented in [39], introducing the motivation for programmable strategies and discussing a number of systems with (some) support for definition of strategies. The essential ingredients of the paradigm of ‘strategic programming’ and their incarnations in other paradigms, such as object-oriented and functional programming, are discussed in [25]. A comparison of strategic programming with adaptive programming is presented in [26]. Finally, the program transformation wiki [1] lists a large number of transformation systems.

## 14 Conclusion

This paper has presented a broad overview of the concepts and applications of the Stratego/XT framework, a language and toolset supporting the high-level implementation of program transformation systems. The framework is applicable to many kinds of transformations, including compilation, generation, analysis, and migration. The framework supports all aspects of program transformation, from the specification of transformation rules, their composition using strategies, to the encapsulation of strategies in tools, and composition of tools into systems.

An important design guideline in Stratego/XT is separation of concerns to achieve reuse at all levels of abstractions. Thus, the separation of rules and strategies allows the specification of rules separately from the strategy that applies them and a generic strategy can be instantiated with different rules. Similarly a certain strategy can be used in different tools, and a tool can be used in different transformation systems. This principle supports reuse of transformations at different levels of granularity.

Another design guideline is that separation of concerns should not draw artificial boundaries. Thus, there is no strict separation between abstraction levels. Rather the distinctions between these levels is methodological and idiomatic rather than semantic. For instance, a rule is really an idiom for a certain type of strategy. Thus, rules and strategies can be interchanged. Similarly, XTC applies strategic control to tools and

allows calling an external tool as though it were a rule. In general, one can mix rules, strategies, tools, and systems as is appropriate for the system under consideration, thus making transformations compositional in practice. Of course one has to consider trade-offs when doing this, e.g., the overhead of calling an external process versus the reuse obtained, but there is no technical objection.

Finally, Stratego/XT is designed and developed as an open language and system. The initial language based on rewriting of abstract syntax trees under the control of strategies has been extended with first-class pattern matching, dynamic rules, concrete syntax, and a tool composition model, in order to address new classes of problems. The library has accumulated many generic transformation solutions. Also the compiler is component-based, and more and more aspects are under the control of the programmer.

Certain aspects of the language could have been developed as a library in a general purpose language. Such an approach, although interesting in its own right, meets with the syntactic and semantic limitations of the host language. Building a domain-specific language for the domain of program transformation has been fruitful. First of all, the constructs that matter can be provided without (syntactic) overhead to the programmer. The separation of concerns (e.g., rules as separately definable entities) that is provided in Stratego is hard to achieve in general purpose languages. Furthermore, the use of the ATerm library with its maximal sharing (hash consing) term model and easy persistence provides a distinct run-time system not available in other languages. Rather than struggling with a host language, the design of Stratego has been guided by the needs of the *transformation* domain, striving to express transformations in a natural way.

Symbolic manipulation and generation of programs is increasingly important in software engineering, and Stratego/XT is an expressive framework for its implementation. The ideas developed in the project can also be useful in other settings. For example, the approach to generic traversal has been transposed to functional, object-oriented, and logic programming [25]. This paper describes Stratego/XT at release 0.9, which is not the final one. There is a host of ideas for improving and extending the language, compiler, library, and support packages, and for new applications. For an overview, see [www.stratego-language.org](http://www.stratego-language.org).

## Acknowledgments

Stratego and XT have been developed with contributions by many people. The initial set of strategy combinators was designed with Bas Luttik. The first prototype language design and compiler was developed with Zino Benaissa and Andrew Tolmach. The run-time system of Stratego is based on the ATerm Library developed at the University of Amsterdam by Pieter Olivier and Hayco de Jong. SDF is maintained and further developed at CWI by Mark van den Brand and Jurgen Vinju. The XT bundle was set up and developed together with Merijn de Jonge and Joost Visser. Martin Bravenboer has played an important role in modernizing XT, collaborated in the development of XTC, and contributed several packages in the area of XML and Java processing. Eelco Dolstra has been very resourceful when it came to compilation and porting issues. The approach to data-flow optimization was developed with Karina Olmos. Rob Vermaas developed the documentation software for Stratego. Many others developed applications or oth-

erwise provided valuable feedback including Otto Skrove Bagge, Arne de Bruijn, Karl Trygve Kalleberg, Dick Kieburtz, Patricia Johann, Lennart Swart, Hedzer Westra, and Jonne van Wijngaarden. Finally, the anonymous referees provided useful feedback on an earlier version of this paper.

## References

1. <http://www.program-transformation.org>.
2. O. S. Bagge, K. T. Kalleberg, M. Haverlaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65–74, Amsterdam, September 2003. IEEE Computer Society Press.
3. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.
4. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *ELAN: User Manual*. Loria, Nancy, France, v3.4 edition, January 27 2000.
5. J. M. Boyle. Abstract programming and program transformation—An approach to reusing programs. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press, 1989.
6. J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transforming system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 353–372. Birkhäuser, 1997.
7. M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software—Practice & Experience*, 30:259–291, 2000.
8. M. G. J. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, April 2003.
9. M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 237–251, Copenhagen, Denmark, July 2002. Springer-Verlag.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
11. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89, Asilomar, Pacific Grove, CA, September 1996. Elsevier.
12. J. R. Cordy, I. H. Carmichael, and R. Halliday. *The TXL Programming Language, Version 8*, April 1995.
13. K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.
14. M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.
15. M. de Jonge. Source tree composition. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *LNCS*, pages 17–32. Springer-Verlag, April 2002.

16. M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proceedings: International Conference on Software Maintenance (ICSM 2001)*, pages 240–249. IEEE Computer Society Press, November 2001.
17. M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001.
18. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996.
19. E. Dolstra and E. Visser. Building interpreters with rewriting strategies. In M. G. J. van den Brand and R. Lämmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science Publishers.
20. B. Fischer and E. Visser. Retrofitting the AutoBayes program synthesis system with concrete syntax. In this volume.
21. M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
22. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
23. P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):1–34, 2000.
24. P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
25. R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming, October 2002. (Draft).
26. R. Lämmel, E. Visser, and J. Visser. Strategic Programming Meets Adaptive Programming. In *Proceedings of Aspect-Oriented Software Development (AOSD'03)*, pages 168–177, Boston, USA, March 2003. ACM Press.
27. B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.
28. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
29. K. Olmos and E. Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.
30. S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.
31. R. J. Rodger. Jostraca: a template engine for generative programming. *European Conference for Object-Oriented Programming*, 2002.
32. L. Swart. Partial evaluation using rewrite rules. A specification of a partial evaluator for Similix in Stratego. Master's thesis, Utrecht University, Utrecht, The Netherlands, August 2002.
33. Terese. *Term Rewriting Systems*. Cambridge University Press, March 2003.
34. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.



35. E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.
36. E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.
37. E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.
38. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
39. E. Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
40. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
41. E. Visser and Z.-e.-A. Benaïssa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, Pont-à-Mousson, France, September 1998. Elsevier Science Publishers.
42. E. Visser, Z.-e.-A. Benaïssa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
43. <http://www.stratego-language.org>.

# Retrofitting the AutoBayes Program Synthesis System with Concrete Syntax

Bernd Fischer<sup>1</sup> and Eelco Visser<sup>2</sup>

<sup>1</sup> RIACS / NASA Ames Research Center, Moffett Field, CA 94035, USA  
fisch@email.arc.nasa.gov

<sup>2</sup> Institute of Information and Computing Sciences, Universiteit Utrecht  
3508 TB Utrecht, The Netherlands  
visser@acm.org

**Abstract.** AUTOBAYES is a fully automatic, schema-based program synthesis system for statistical data analysis applications. Its core component is a schema library, i.e., a collection of generic code templates with associated applicability constraints which are instantiated in a problem-specific way during synthesis. Currently, AUTOBAYES is implemented in Prolog; the schemas thus use abstract syntax (i.e., Prolog terms) to formulate the templates. However, the conceptual distance between this abstract representation and the concrete syntax of the generated programs makes the schemas hard to create and maintain.

In this paper we describe how AUTOBAYES is retrofitted with concrete syntax. We show how it is integrated into Prolog and describe how the seamless interaction of concrete syntax fragments with AUTOBAYES's remaining "legacy" meta-programming kernel based on abstract syntax is achieved. We apply the approach to gradually migrate individual schemas without forcing a disruptive migration of the entire system to a different meta-programming language. First experiences show that a smooth migration can be achieved. Moreover, it can result in a considerable reduction of the code size and improved readability of the code. In particular, abstracting out fresh-variable generation and second-order term construction allows the formulation of larger continuous fragments.

## 1 Introduction

Program synthesis and transformation systems work on two language levels, the object-level (i.e., the language of the manipulated programs), and the meta-level (i.e., the implementation language of the system itself). Conceptually, these two levels are unrelated but in practice they have to be interfaced with each other. Often, the object-language is simply embedded within the meta-language, using a data type to represent the abstract syntax trees of the object-language. Although the actual implementation mechanisms (e.g., records, objects, or algebraic data types) may vary, embeddings can be used with essentially all meta-languages, making their full programming capabilities immediately available for program manipulations. Meta-level representations of object-level program fragments are then constructed in an essentially syntax-free fashion, i.e., not using the notation of the object-language, but using the operations provided by the data type.

However, syntax matters. The conceptual distance between the concrete programs that we understand and the meta-level representations that we need to use grows with the complexity of the object-language syntax and the size of the represented program fragments, and the use of abstract syntax becomes less and less satisfactory. Languages like Prolog and Haskell allow a rudimentary integration of concrete syntax via user-defined operators. However, this is usually restricted to simple precedence grammars, entailing that realistic object-languages cannot be represented well if at all. Traditionally, a quotation/anti-quotation mechanism is thus used to interface languages: a quotation denotes an object-level fragment, an anti-quotation denotes the result of a meta-level computation which is spliced into the object-level fragment. If object-language and meta-language coincide, the distinction between the language levels is purely conceptual, and switching between the levels is easy; a single compiler can be used to process both levels. If the object-language is user-definable, the mechanism becomes more complicated to implement and usually requires specialized meta-languages such as ASF+SDF [5], Maude [3], or TXL [4] which support syntax definition and reflection.

AUTOBAYES [9, 8] is a program synthesis system for the statistical data analysis domain. It is a large and complex software system implemented in Prolog and its complexity is comparable to a compiler. The synthesis process is based on schemas which are written in Prolog and use abstract syntax representations of object-program fragments. The introduction of concrete syntax would simplify the creation and maintenance of these schemas. However, a complete migration of the implementation of AUTOBAYES to a different meta-programming language requires a substantial effort and disrupts the ongoing system development. To avoid this problem, we have chosen a different path.

In this chapter, we thus describe the first experiences with our ongoing work on adding support for user-definable concrete syntax to AUTOBAYES. We follow the general approach outlined in [15], which allows the extension of an arbitrary meta-language with concrete object-language syntax by combining the syntax definitions of both languages. We show how the approach is instantiated for Prolog and describe the processing steps required for a seamless interaction of concrete syntax fragments with the remaining “legacy” meta-programming system based on abstract syntax—despite all its idiosyncrasies. With this work we show that the approach of [15] can indeed be applied to meta-languages other than Stratego. To reduce the effort of making such instantiations we have constructed a generic tool encapsulating the process of parsing a program using concrete object-syntax. Furthermore, we have extended the approach with object-level comments, and object-language specific transformations for integrating object-level abstract syntax in the meta-language.

The original motivation for this specific path was purely pragmatic. We wanted to realize the benefits of concrete syntax without forcing the disruptive migration of the entire system to a different meta-programming language. Retrofitting Prolog with support for concrete syntax allows a gradual migration. Our long-term goal, however, is more ambitious: we want to support domain experts in creating and maintaining schemas. We expect that the use of concrete syntax makes it easier to gradually “schematize” existing domain programs. We also plan to use different grammars to describe programs on different levels of abstraction and thus to support domain engineering.

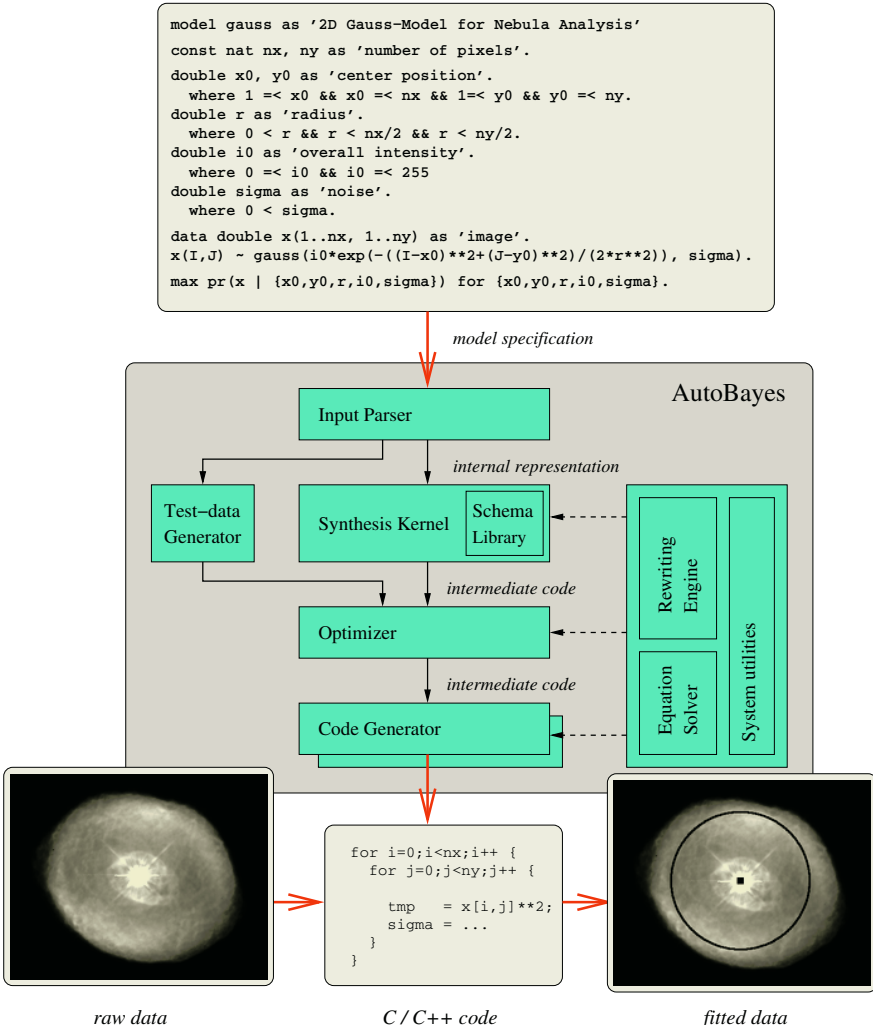


Fig. 1. AUTOBAYES system architecture.

## 2 Overview of the AutoBayes System

AUTOBAYES is a fully automatic program synthesis system for data analysis problems. It has been used to derive programs for applications like the analysis of planetary nebulae images taken by the Hubble space telescope [7, 6] as well as research-level machine learning algorithms [1]. It is implemented in SWI-Prolog [18] and currently comprises about 75,000 lines of documented code; Figure 1 shows the system architecture.

AUTOBAYES derives code from a *statistical model* which describes the expected properties of the data in a fully declarative fashion: for each problem variable (i.e., observation or parameter), properties and dependencies are specified via probability

distributions and constraints. The top box in Figure 1 shows the specification of a nebulae analysis model. The last two clauses are the core of this specification; the remaining clauses declare the model constants and variables, and impose constraints on them. The distribution clause

$$x(I, J) \sim \text{gauss}(i_0 * \exp(-((I-x_0)**2+(J-y_0)**2)/(2*r**2)), \text{sigma}).$$

states that, with an expected error *sigma*, the expected value of the observation *x* at a given position (*i, j*) is a function of this position and the nebula's unknown center position (*x<sub>0</sub>, y<sub>0</sub>*), radius *r*, and overall intensity *i<sub>0</sub>*. The task clause

$$\max \text{pr}(x | \{i_0, x_0, y_0, r, \text{sigma}\}) \text{ for } \{i_0, x_0, y_0, r, \text{sigma}\}.$$

specifies the analysis task, which the synthesized program has to solve, i.e., to estimate the parameter values which maximize the probability of actually observing the given data and thus under the given model best explain the observations. In this case, the task can be solved by a mean square error minimization due to the gaussian distribution of the data and the specific form of the probability. Note, however, that (i) this is not immediately clear from the model, (ii) the function to be minimized is not explicitly given in the model, and (iii) even small modifications of the model may require completely different algorithms.

AUTOBAYES derives the code following a schema-based approach. A *program schema* consists of a parameterized code fragment (i.e., template) and a set of constraints. Code fragments are written in ABIR (AUTOBAYES Intermediate Representation), which is essentially a “sanitized” variant of C (e.g., neither pointers nor side effects in expressions) but also contains a number of domain-specific constructs (e.g., vector/matrix operations, finite sums, and convergence-loops). The fragments also contain parameterized object-level comments which eventually become the documentation of the synthesized programs. The parameters are instantiated either directly by the schema or by AUTOBAYES calling itself recursively with a modified problem. The constraints determine whether a schema is applicable and how the parameters can be instantiated. They are formulated as conditions on the model, either directly on the specification, or indirectly on a Bayesian network [2] extracted from the specification. Such networks are directed, acyclic graphs whose nodes represent the variables specified in the model and whose edges represent the probabilistic dependencies between them, as specified by the distribution clauses: the variable on the left-hand side depends on all model variables occurring on the right-hand side. In the example, each  $x_{ij}$  thus depends on  $i_0$ ,  $x_0$ ,  $y_0$ ,  $r$  and *sigma*.

The schemas are organized hierarchically into a schema library. Its top layers contain decomposition schemas based on independence theorems for Bayesian networks which try to break down the problem into independent sub-problems. These are domain-specific divide-and-conquer schemas: the emerging sub-problems are fed back into the synthesis process and the resulting programs are composed to achieve a solution for the original problem. Guided by the network structure, AUTOBAYES is thus able to synthesize larger programs by composition of different schemas. The core layer of the library contains statistical algorithm schemas as for example *expectation maximization* (EM) [10] and *nearest neighbor clustering* (k-Means); usually, these generate the skeleton of

the program. The final layer contains standard numeric optimization methods as for example the simplex method or different conjugate gradient methods. These are applied after the statistical problem has been transformed into an ordinary numeric optimization problem and AUTOBAYES failed to find a symbolic solution for that problem. The schemas in the upper layers of the library are very similar to the underlying theorems and thus contain only relatively small code fragments while the schemas in the lower layers closely resemble “traditional” generic algorithm templates. Their code fragments are much larger and make full use of ABIR’s language constructs. These schemas are the focus of our migration approach.

The schemas are applied exhaustively until all maximization tasks are rewritten into ABIR code. The schemas can explicitly trigger large-scale optimizations which take into account information from the synthesis process. For example, all numeric optimization routines restructure the goal expression using code motion, common sub-expression elimination, and memoization. In a final step, AUTOBAYES translates the ABIR code into code tailored for a specific run-time environment. Currently, it provides code generators for the Octave and Matlab environments; it can also produce standalone C and Modula-2 code. The entire synthesis process is supported by a large meta-programming kernel which includes the graphical reasoning routines, a symbolic-algebraic subsystem based on a rewrite engine, and a symbolic equation solver.

### 3 Program Generation in Prolog

In the rest of this chapter we will describe how AUTOBAYES is retrofitted with concrete object syntax for the specification of program schemas. We start in this section with a description of program generation with abstract syntax in Prolog. In Section 4 we describe the replacement of abstract syntax with concrete syntax. In Sections 5 to 7 we then discuss the techniques used to implement this embedding.

Figure 2 shows an excerpt of a schema that implements (i.e., generates code for) the Nelder-Mead simplex method for numerically optimizing a function with respect to a set of variables [11]. The complete schema comprises 508 lines of documented Prolog-code, and is fairly typical in most aspects, e.g., the size of the overall schema and of the fragment, respectively, the amount of meta-programming, or the ratio between the code constructed directly (e.g., *Code*) and recursively (e.g., *Reflection*). This schema is also used to generate the algorithm core for the nebula specification example.

#### 3.1 Abstract Syntax in Prolog

The simplex schema is implemented as a single Prolog clause which takes as arguments an expression *Formula* representing the function to be optimized, a set *Vars* of target variables, and an expression *Constraint* representing the constraints on all variables. It returns as result *Code* an ABIR code fragment which contains the appropriately instantiated Nelder-Mead simplex algorithm. This code is represented by means of a Prolog term. In general, a Prolog term is either an atom, a variable<sup>1</sup>, a *functor* application  $f(t_1, \dots, t_n)$ , applying a functor  $f$  to Prolog terms  $t_i$ , or a list  $[t_1, \dots, t_n]$

<sup>1</sup> Prolog uses capitalization to distinguish a variable  $X$  from an atom  $x$ .

```

schema(Formula, Vars, Constraint, Code) :-
  ...
  model_gensym(simplex, Simplex),
  SDim = [dim(A_BASE, Size1), dim(A_BASE, Size0)],
  SDecl = matrix(Simplex, double, SDim,
    [comment(['Simplex data structure: (', Size, '+1) ',
      'points in the ', Size,
      '-dimensional space'])]),
  ...
  var_fresh(I),
  var_fresh(J),
  index_make([I, dim(A_BASE, Size0)], Index_i),
  index_make([J, dim(A_BASE, Size1)], Index_j),
  Center_i =.. [Center, I],
  Simplex_ji =.. [Simplex, J, I],
  Centroid =
    for([Index_i],
      assign(Center_i, sum([Index_j], Simplex_ji), []),
      [comment(['Calculate the center of gravity in the simplex'])]),
  ...
  simplex_try(Formula, Simplex, ...,
    -1, 'Reflect the simplex from the worst point (F = -1)',
    Reflection),
  ...
  Loop = while(converging([...]),
    series([Centroid, Reflection, ...], []),
    [comment('Convergence loop')]),
  ...
  Code = block(local([SDecl, ...]),
    series([Init, Loop, Copy], []),
    [label(SLabel), comment(XP)]).

```

**Fig. 2.** AUTOBAYES-schema for the Nelder-Mead simplex method (excerpt).

of terms. An ABIR program is then represented as a term by using a functor for each construct in the language, for example:

```

assign : lvalue * expression * list(comment) -> statement
for    : list(index) * statement * list(comment) -> statement
series : list(statement) * list(comment) -> statement
sum    : list(index) * expression -> expression

```

Thus, if  $i$  represents an index,  $s$  a statement, and  $c$  a comment, the term `for([i], s, [c])` represents a for-statement.

Each goal of the form  $X = t$  binds a Prolog term  $t$  to a Prolog variable  $X$ . However, in the schema, the program for `Code` is not constructed as a single large term, but rather assembled from smaller fragments by including the terms bound to these variables. In Figure 2, the terms corresponding to ABIR fragments are distinguished typographically using italics, but this is a conceptual distinction only.

### 3.2 Meta-programming Kernel

In addition to goals composing program fragments by direct term formation, the schema contains recursive schema invocations such as `simplex_try`, which produces code for the *Reflection* fragment from a more constrained version of *Formula*. Furthermore, the schema calls a number of meta-programming predicates. For example, the `var_fresh(X)` predicate generates a fresh object variable and binds it to its argument, which is a meta-level variable. This prevents variable clashes in the generated program. Similarly, the `index_make` predicate constructs an index expression.

The schema in Figure 2 also uses second-order terms to represent array accesses or function calls where the names are either given as parameters or automatically renamed apart from a meaningful root (cf. the `model_gensym(simplex, Simplex)` goal). A fully abstract syntax would use additional functors for these constructs and represent for example an access to the array `simplex` with subscripts `pv0` and `pv1` by `arraysub(simplex, [var(pv0), var(pv1)])`. However, this makes the abstract syntax rather unwieldy and much harder to read. Therefore, such constructs are abbreviated by means of simple functor applications, e.g., `simplex(pv0, pv1)`. Unfortunately, Prolog does not allow second-order term formation, i.e., terms with variables in the functor-position. Instead, it is necessary to use the built-in `=..`-operator, which constructs a functor application from a list where the head element is used as functor name and the rest of the list contains the arguments of the application. Hence, the schemas generate array access expressions such as the one above by goals such as `Simplex_ji =.. [Simplex, J, I]`, where the meta-variables `Simplex`, `J`, and `I` are bound to concrete names.

## 4 Migrating from Abstract Syntax to Concrete Syntax

The excerpt shows why the simple abstract syntax approach quickly becomes cumbersome as the schemas become larger. The code fragment is built up from many smaller fragments by the introduction of new meta-variables (e.g., `Loop`) because the abstract syntax would become unreadable otherwise. However, this makes it harder to follow and understand the overall structure of the algorithm. The schema is sprinkled with a large number of calls to small meta-programming predicates, which makes it harder to write schemas because one needs to know not only the abstract syntax, but also a large part of the meta-programming base. In our experience, these peculiarities make the learning curve much steeper than it ought to be, which in turn makes it difficult for a domain expert to gradually extend the system's capabilities by adding a single schema.

In the following, we illustrate how this schema is migrated and refactored to make use of concrete syntax, using the *Centroid* fragment as running example.

### 4.1 Concrete Syntax

The first step of the migration is to replace terms representing program fragments in abstract syntax by the equivalent fragments in the concrete syntax of ABIR. Thus, the *Centroid* fragment becomes:



```
Centroid = |[
  /* Calculate the center of gravity in the simplex */
  for( Index_i:idx )
    Center_i := sum( Index_j:idx ) Simplex_ji:exp
]|
```

Here, we use `| [ ... ] |` to quote a piece of ABIR code within a Prolog program.

## 4.2 Meta-variables

In the translation to concrete syntax, Prolog variables in a term are *meta-variables*, i.e., variables ranging *over* ABIR code, rather than variables *in* ABIR code. In the fragment `| [ x := 3 + j ] |`, `x` and `j` are ABIR variables, whereas in the fragment `| [ x := 3 + J:exp ] |`, `x` is an ABIR variable, but `J:exp` is a meta-variable ranging over expressions. For the embedding of ABIR in Prolog we use the convention that meta-variables are distinguished by capitalization and can thus be used directly in the concrete syntax without tags. In a few places, the meta-variables are tagged with their syntactic category, e.g., `Index_i:idx`. This allows the parser to resolve ambiguities and to introduce the injection functions necessary to build well-formed syntax trees.

## 4.3 Abstracting from Meta-programming Operations

The next migration step eliminates calls to meta-programming predicates and replaces them by appropriate abstractions in ABIR. First, we remove the creation of index expressions such as `index_make([I, dim(A_BASE, Size0)], Index_i)` and replace the corresponding `Index` variables directly with the more natural loop index notation:

```
Centroid = |[
  /* Calculate the center of gravity in the simplex */
  for( I := A_BASE .. Size0 )
    Center_i := sum( J := A_BASE .. Size1 ) Simplex_ji:exp
]|
```

Incidentally, this also eliminates the need for the `idx`-tags because the syntactic category is now determined by the source text.

Next, array-reference creation with the `=..` operator is replaced with array access notation in the program fragment:

```
Centroid = |[
  /* Calculate the center of gravity in the simplex */
  for( I := A_BASE .. Size0 )
    Center[I] := sum( J := A_BASE .. Size1 ) Simplex[J, I]
]|
```

Finally, the explicit generation of fresh object-variables using `var_fresh` is expressed *in* the code by tagging the corresponding meta-variable with `@new`, a special anti-quotation operator which constructs fresh object-level variable names.

```
Centroid = |[
  /* Calculate the center of gravity in the simplex */
  for( I@new := A_BASE .. Size0 )
    Center[I] := sum( J@new := A_BASE .. Size1 ) Simplex[J, I]
]|
```

Thus 10 lines of code have been reduced to 5 lines, which are more readable.

#### 4.4 Fragment Inlining

The final step of the migration consists of refactoring the schema by inlining program fragments; the fragments are self-descriptive, do not depend on separate calls to meta-programming predicates, and can be read as pieces of code. For example, the fragment for `Centroid` above can be inlined in the fragment for `Loop`, which itself can be inlined in the final `Code` fragment. After this refactoring, a schema consists of one, or a few, large program patterns.

In the example schema the use of concrete syntax, `@new`, and inlining reduces the overall size by approximately 30% and eliminates the need for explicit meta-programming. The reduction ratio is more or less maintained over the entire schema. After migration along the lines above, the schema size is reduced from 508 lines to 366 lines. After white space removal, the original schema contains 7779 characters and the resulting schema with concrete syntax 5538, confirming a reduction of 30% in actual code size. At the same time, the resulting fewer but larger code fragments give a better insight into the structure of the generated code.

### 5 Embedding Concrete Syntax into Prolog

The extension of Prolog with concrete syntax as sketched in the previous section is achieved using the syntax definition formalism SDF2 [14, 12] and the transformation language Stratego [16, 13] following the approach described in [15]. SDF is used to specify the syntax of ABIR and Prolog as well as the embedding of ABIR into Prolog. Stratego is used to transform syntax trees over this combined language into a pure Prolog program. In this section we explain the syntactical embedding, and in the next two sections we outline the transformations mapping Prolog with concrete syntax to pure Prolog.

#### 5.1 Syntax of Prolog and ABIR

The extension of a meta-language with concrete object syntax requires an embedding of the syntax of object code fragments as expressions in the meta-language. We thus created syntax definitions for Prolog and ABIR using SDF. An SDF production  $A_1 \dots A_n \rightarrow A_0$  is a context-free grammar rule that declares that the concatenation of strings of sorts  $A_1$  to  $A_n$  is a string of sort  $A_0$ . The following is a fragment from the Prolog syntax definition with productions for clauses and terms. Note that the SDF construct  $\{S \ 1\}^+$  denotes one or more  $S$ s *separated by 1*s.

```

module Prolog
exports
  context-free syntax
    Head ":-" Body "." -> Clause {cons("nonunitclause")}
    Goal                -> Body {cons("bodygoal")}
    Term                -> Goal
    Functor "(" {Term ","}+ ")" -> Term {cons("func")}
    Term Op Term        -> Term {cons("infix")}
    Variable            -> Term {cons("var")}
    Atom                -> Term {cons("atom")}
    Name                -> Functor {cons("functor")}
    Name                -> Op {cons("op")}

```

The  $\{ \text{cons}(c) \}$  annotations in the productions declare the constructors to be used in abstract syntax trees corresponding to the parse trees over the syntax definition. Similarly, the following is a fragment from the syntax definition of ABIR:

```

module ABIR
exports
  context-free syntax
    LValue "!=" Exp -> Stat {cons("assign")}
    "for" "(" IndexList ")" Stat -> Stat {cons("for")}
    {Index ","}* -> IndexList {cons("indexlist")}
    Id "!=" Exp ".." Exp -> Index {cons("index")}

```

## 5.2 Combining Syntax Definitions

Since SDF is a modular syntax definition formalism, combining languages is simply a matter of importing the appropriate modules. In addition, object-language expressions should be embedded in meta-language expressions. The following module defines such an embedding of ABIR into Prolog:

```

module PrologABIR
imports Prolog ABIR
exports
  context-free syntax
    "[" Exp "]" -> Term {cons("toterm")}
    "[" Stat "]" -> Term {cons("toterm")}
  variables
    [A-Z][A-Za-z0-9_]* -> Id {prefer}
    [A-Z][A-Za-z0-9_]* ":"exp" -> Exp

```

The module declares that ABIR Expressions and Statemements can be used as Prolog terms by quoting them with the `[ ]` delimiters, as we have seen in the previous section. The `variables` section declares schemas for *meta-variables*. Thus, a capitalized identifier can be used as a meta-variable for identifiers, and a capitalized identifier tagged with `:exp` can be used as a meta-variable for expressions.

## 6 Exploding Embedded Abstract Syntax

### 6.1 Embedded Abstract Syntax

After parsing a schema with the combined syntax definition the resulting abstract syntax tree is a mixture of Prolog and ABIR abstract syntax. For example, the Prolog-goal

```
Code = |[ X := Y:exp + z ]|
```

is parsed into the abstract syntax term

```
bodygoal(infix(var("Code"), op(symbol("=")),
               toterm(assign(var(meta-var("X")),
                             plus(meta-var("Y:exp"), var("z"))))))
```

The language transitions are characterized by the `toterm`-constructor, and meta-variables are indicated by the `meta-var`-constructor. Thus, `bodygoal` and `infix` belong to Prolog abstract syntax, while `assign`, `var` and `plus` belong to ABIR abstract syntax.

### 6.2 Exploding

A mixed syntax tree can be translated to a pure Prolog tree by “exploding” embedded tree constructors to functor applications<sup>2</sup>:

```
bodygoal(infix(var("Code"), op(symbol("=")),
               func(functor(word("assign")),
                    [func(functor(word("var")), [var("X")]),
                     func(functor(word("plus")),
                          [var("Y:exp"),
                           func(functor(word("var")),
                                [atom(quotedname("'z'"))])])])]))
```

After pretty-printing this tree we get the pure Prolog-goal

```
Code = assign(var(X), plus(Y, var('z')))
```

Note how the meta-variables `X` and `Y` have become Prolog variables representing a variable name and an expression, respectively, while the object variable `z` has become a character literal. Also note that `X` is a meta-variable for an *object-level identifier* and will eventually be instantiated with a character literal, while `Y` is a variable for an *expression*.

### 6.3 Implementing Explosion in Stratego

Explosion is defined generically using transformations on mixed syntax trees, i.e., it is independent from the object language. The complete explosion transformation takes about 35 lines of Stratego and deals with special cases such as strings and lists, but the

---

<sup>2</sup> Note that `functor` is just a term label and different from the built-in predicate `functor/3`.

```

strategies
  explode = alltd(?toterm(<trm-explode>))
  trm-explode = trm-metavar <+ trm-op
rules
  trm-metavar : meta-var(X) -> var(X)
  trm-op : Op#([]) -> atom(word(<lower-case>Op))
  trm-op : Op#([T | Ts]) -> func(funcutor(word(<lower-case>Op)),
                                <map(trm-explode)>[T | Ts])

```

**Fig. 3.** Rules and strategy for exploding embedded abstract syntax.

essence of the transformation is shown in Figure 3. A detailed explanation of the specification is beyond the scope of this chapter. For an introduction to Stratego see [13].

The `explode` strategy uses the generic traversal strategy `alltd` to descend into the abstract syntax tree of the Prolog program. When encountering a term constructed with `toterm`, its argument is exploded using the `trm-explode` transformation, which either applies one of the rules `trm-op` or the rule `trm-metavar`. The latter rule turns a meta-variable encountered in an embedded term into a Prolog variable. The `trm-op` rules transform constructor applications. The left-hand side of the rules have the form `Op#(Ts)`, thus generically decomposing a constructor application into its constructor (or operator) `Op`, and the list of arguments `Ts`. If the list of arguments is empty, an atom is produced. Otherwise a functor application is produced, where the arguments of the functor are recursively exploded by mapping the `trm-explode` strategy over the list of arguments.

## 7 Custom Abstract Syntax

Parsing and then exploding the final Centroid-fragment on page 247 then produces the pure Prolog-goal

```

Centroid =
  commented(
    comment(['Calculate the center of gravity in the simplex ']),
    for(indexlist([index(newvar(I),var(A_BASE),var(Size0))]),
      assign(arraysub(Center,[var(I)]),
        sum(indexlist([index(newvar(J),
          var(A_BASE),var(Size1))]),
          call(Simplex,[var(J),var(I)]))))))

```

Comparing the generated Centroid-goal above with the original in Figure 2 shows that the abstract syntax underlying the concrete syntax fragments does not correspond exactly to the original abstract syntax used in AutoBayes. That is, two different abstract syntax formats are used for the ABIR language. The format used in AUTOBAYES (e.g., Figure 2) is less explicit since it uses Prolog functor applications to represent array references and function calls, instead of the more verbose representation underlying the concrete syntax fragments.

In order to interface schemas written in concrete syntax with legacy components of the synthesis system, additional transformations are applied to the Prolog code, which translate between the two versions of the abstract syntax. For the Centroid-fragment this produces:

```
Centroid =
  for([idx(newvar(I),A_BASE,Size0)],
    assign(arraysub(Center,[I]),
      sum([idx(newvar(J),A_BASE,Size1)],call(Simplex,[J,I]))),
    [comment(['Calculate the center of gravity in the simplex '])])
```

## 7.1 Lifting Predicates

In AutoBayes, array accesses are represented by means of functor applications and object variable names are generated by gensym-predicates. This cannot be expressed in a plain Prolog term. Thus arraysubs and calls are hoisted out of abstract syntax terms and turned into term constructors and fresh variable generators as follows:

```
var_fresh(I), _a =.. [Center,I], var_fresh(J), _b =.. [Simplex,J,I],
Centroid =
  for([idx(I, A_BASE, Size0)],
    assign(_a, sum([idx(J, A_BASE, Size1)], _b)),
    [comment(['Calculate the center of gravity in the simplex '])])
```

Hence, the embedded concrete syntax is transformed exactly into the form needed to interface it with the legacy system.

## 8 Conclusions

Program generation and transformation systems manipulate large, parameterized object language fragments. Operating on such fragments using abstract-syntax trees or string-based concrete syntax is possible, but has severe limitations in maintainability and expressive power. Any serious program generator should thus provide support for concrete object syntax together with the underlying abstract syntax.

In this chapter we have shown that the approach of [15] can indeed be generalized to meta-languages other than Stratego and that it is thus possible to add such support to systems implemented in a variety of meta-languages. We have applied this approach to AutoBayes, a large program synthesis system that uses a simple embedding of its object-language (ABIR) into its meta-language (Prolog). The introduction of concrete syntax results in a considerable reduction of the schema size ( $\approx 30\%$ ), but even more importantly, in an improved readability of the schemas. In particular, abstracting out fresh-variable generation and second-order term construction allows the formulation of larger continuous fragments and improves the locality in the schemas. Moreover, meta-programming with concrete syntax is cheap: using Stratego and SDF, the overall effort to develop all supporting tools was less than three weeks. Once the tools were in place, the migration of a schema was a matter of a few hours. Finally, the experiment has also demonstrated that it is possible to introduce concrete syntax support gradually, without

forcing a disruptive migration of the entire system to the extended meta-language. The seamless integration with the “legacy” meta-programming kernel is achieved with a few additional transformations, which can be implemented quickly in Stratego.

## 8.1 Contributions

The work described in this chapter makes three main contributions to domain-specific program generation. First, we described an extension of Prolog with concrete object syntax, which is a useful tool for all meta-programming systems using Prolog. The tools that implement the mapping back into pure Prolog are available for embedding arbitrary object languages into Prolog<sup>3</sup>. Second, we demonstrated that the approach of [15] can indeed be applied to meta-languages other than Stratego. We extended the approach by incorporating concrete syntax for object-level comments and annotations, which are required for documentation and certification of the generated code [17]. Third, we also extended the approach with object-language-specific transformations to achieve a seamless integration with the legacy meta-programming kernel. This allows a gradual migration of existing systems, even if they were originally designed without support for concrete syntax in mind. These transformations also lift meta-computations from object code into the surrounding meta-code. This allows us to introduce abstractions for fresh variable generation and second-order variables to Prolog.

## 8.2 Future Work

In future work, we will migrate more schemas to concrete syntax to make the maintenance of the AUTOBAYES easier. We expect that these changes will confirm our estimate of 30% reduction in the size of the schemas.

We also plan to investigate the usefulness of concrete syntax in a gradual “schematization” of existing domain programs. The basic idea here is to use the existing program initially unmodified as code fragment in a very specialized schema, and then to abstract it incrementally, e.g., by parameterizing out names or entire computations which can then be re-instantiated differently during synthesis. Finally, we plan to use grammars as types to enforce that the fragments are not only syntactically well-formed but actually contain code of the right form. We hope that we can support domain engineering by using grammars on different levels of abstraction.

## Acknowledgements

We would like to thank the anonymous referees for their comments on a previous version of this paper.

## References

1. W. Buntine, B. Fischer, and A. G. Gray. Automatic derivation of the multinomial PCA algorithm. Technical report, NASA/Ames, 2003. Available at <http://ase.arc.nasa.gov/people/fischer/>.

---

<sup>3</sup> <http://www.stratego-language.org/Stratego/PrologTools>

2. W. L. Buntine. Operations for learning with graphical models. *JAIR*, 2:159–225, 1994.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
4. J. R. Cordy, I. H. Carmichael, and R. Halliday. *The TXL Programming Language, Version 8*, April 1995.
5. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996.
6. B. Fischer, A. Hajian, K. Knuth, and J. Schumann. Automatic derivation of statistical data analysis algorithms: Planetary nebulae and beyond. Technical report, NASA/Ames, 2003. Available at <http://ase.arc.nasa.gov/people/fischer/>.
7. B. Fischer and J. Schumann. Applying autobayes to the analysis of planetary nebulae images. In J. Grundy and J. Penix, editors, *Proc. 18th ASE*, pages 337–342, Montreal, Canada, October 6–10 2003. IEEE Comp. Soc. Press.
8. B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *JFP*, 13(3):483–508, May 2003.
9. A. G. Gray, B. Fischer, J. Schumann, and W. Buntine. Automatic derivation of statistical algorithms: The EM family and beyond. In S. Becker, S. Thrun, and K. Obermayer, editors, *NIPS 15*, pages 689–696. MIT Press, 2003.
10. G. McLachlan and T. Krishnan. *The EM Algorithm and Extensions*. Wiley Series in Probability and Statistics. John Wiley & Sons, New York, 1997.
11. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge Univ. Press, Cambridge, UK, 2nd. edition, 1992.
12. M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.
13. E. Visser. Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9. In this volume.
14. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
15. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
16. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
17. M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. In L.-H. Eriksson and P. A. Lindsay, editors, *Proc. FME 2002: Formal Methods—Getting IT Right*, volume 2391 of *LNCS*, pages 431–450, Copenhagen, Denmark, July 2002. Springer.
18. J. Wielemaker. *SWI-Prolog 5.2.9 Reference Manual*. Amsterdam, 2003.



# Optimizing Sequences of Skeleton Calls

Herbert Kuchen

University of Münster, Department of Information Systems  
Leonardo Campus 3, D-48159 Münster, Germany  
`kuchen@uni-muenster.de`

**Abstract.** Today, parallel programming is dominated by message passing libraries such as MPI. Algorithmic skeletons intend to simplify parallel programming by their expressive power. The idea is to offer typical parallel programming patterns as polymorphic higher-order functions which are efficiently implemented in parallel. Skeletons can be understood as a domain-specific language for parallel programming. In this chapter, we describe a set of data parallel skeletons in detail and investigate the potential of optimizing sequences of these skeletons by replacing them by more efficient sequences. Experimental results based on a draft implementation of our skeleton library are shown.

## 1 Introduction

Today, parallel programming of MIMD machines with distributed memory is typically based on message passing. Owing to the availability of standard message passing libraries such as MPI<sup>1</sup> [GL99], the resulting software is platform independent and efficient. However, the programming level is still rather low and programmers have to fight against low-level communication problems such as deadlocks. Moreover, the program is split into a set of processes which are assigned to the different processors. Like an ant, each process only has a local view of the overall activity. A global view of the overall computation only exists in the programmer's mind, and there is no way to express it more directly on this level.

Many approaches try to increase the level of parallel programming and to overcome the mentioned disadvantages. Here, we will focus on *algorithmic skeletons*, i.e. typical parallel programming patterns which are efficiently implemented on the available parallel machine and usually offered to the user as higher-order functions, which get the details of the specific application problem as argument functions (see e.g. [Co89,BK98,DP97]).

Object-oriented programmers often simulate higher-order functions using the “template” design pattern [GH95], where the general algorithmic structure is defined in an abstract superclass, while its subclasses provide the application-specific details by implementing certain template methods appropriately. In our approach, there is no need for static subclasses and the corresponding syntactic

---

<sup>1</sup> We assume some familiarity with MPI and C++.

overhead for providing them, but these details are just passed as arguments. Another way of simulating higher-order functions in an object-oriented language is to use the “command” design pattern [GH95]. Here, the “argument functions” are encapsulated by command objects. But still this pattern causes substantial overhead.

In our framework, a parallel computation consists of a sequence of calls to skeletons, possibly interleaved by some local computations. The computation is now seen from a global perspective. As explained in [Le04], skeletons can be understood as a domain-specific language for parallel programming. Several implementations of algorithmic skeletons are available. They differ in the kind of host language used and in the particular set of skeletons offered. Since higher-order functions are taken from functional languages, many approaches use such a language as host language [Da93,KP94,Sk94]. In order to increase the efficiency, imperative languages such as C and C++ have been extended by skeletons, too [BK96,BK98,DP97,FO92].

Depending on the kind of parallelism used, skeletons can be classified into *task parallel* and *data parallel* ones. In the first case, a skeleton (dynamically) creates a system of communicating processes by nesting predefined process topologies such as **pipeline**, **farm**, **parallel composition**, **divide&conquer**, and **branch&bound** [DP97,Co89,Da93,KC02]. In the second case, a skeleton works on a distributed data structure, performing the same operations on some or all elements of this data structure. Data-parallel skeletons, such as **map**, **fold** or **rotate** are used in [BK96,BK98,Da93,Da95,DP97,KP94].

Moreover, there are implementations offering skeletons as a library rather than as part of a new programming language. The approach described in the sequel is based on the skeleton library introduced in [Ku02,KC02,KS02] and on the corresponding C++ language binding. As pointed out by Smaragdakis [Sm04], C++ is particularly suited for domain-specific languages due to its meta-programming abilities. Our library provides task as well as data parallel skeletons, which can be combined based on the *two-tier model* taken from P<sup>3</sup>L [DP97]. In general, a computation consists of nested task parallel constructs where an atomic task parallel computation can be sequential or data parallel. Purely data parallel and purely task parallel computations are special cases of this model. An advantage of the C++ binding is that the three important features needed for skeletons, namely higher-order functions (i.e. functions having functions as arguments), partial applications (i.e. the possibility to apply a function to less arguments than it needs and to supply the missing arguments later), and parametric polymorphism, can be implemented elegantly and efficiently in C++ using operator overloading and templates, respectively [St00,KS02].

Skeletons provide a global view of the computation which enables certain optimizations. In the spirit of the well-known Bird-Meertens formalism [Bi88,Bi89][GL97], algebraic transformations on sequences of skeleton calls allow one to replace a sequence by a semantically equivalent but more efficient sequence. The investigation of such sequences of skeletons and their transformation will be the core of the current chapter. Similar transformations can be found in

[Go04, GL97, GW99, BG02]. There, you can also find correctness proofs of the transformations. All these transformations are expressed in terms of classical map, fold, and scan operations on lists, abstracting from data-distribution issues. These transformations are very elegant, but unfortunately exchanging lists between processors causes a substantial overhead for marshaling and unmarshaling. For this reason, we consider skeletons working on distributed arrays and matrices rather than on lists or sequences of values. Arrays and matrices are data structures frequently used in parallel programming, since their partitions can be easily shipped to other processors and since their elements can be efficiently accessed. Thus, skeletons working on them are particularly important in practice. Our skeletons explicitly control the data distribution, giving the programmer a tighter control of the computation. Moreover, our skeletons partly update an array or matrix in place. This is typically more efficient than generating a new data structure like the usual and well-known skeletons for lists do it. Updates in place are possible, since we are in an imperative / object oriented setting rather than in a functional language.

This chapter is organized as follows. In Section 2, we present the main concepts of our skeleton library and explain its skeletons in detail. In Section 3, we investigate the optimization of sequences of these skeletons and show some experimental results. Finally, in Section 4 we conclude.

## 2 The Skeleton Library

The skeleton library offers data parallel and task parallel skeletons. Data parallelism is based on a *distributed data structure*, which is manipulated by operations processing it as a whole and which happen to be implemented in parallel internally. Task parallelism is established by setting up a system of processes which communicate via streams of data. Such a system is not arbitrarily structured but constructed by nesting predefined process topologies such as farms and pipelines. Moreover, it is possible to nest task and data parallelism according to the mentioned two-tier model of P<sup>3</sup>L, which allows atomic task parallel processes to use data parallelism inside. Here, we will focus on data parallelism and on the optimization of sequences of data parallel skeletons. Details on task parallel skeletons can be found in [KC02].

### 2.1 Overview of Data Parallel Skeletons

As mentioned above, data parallelism is based on a *distributed data structure* (or several of them). This data structure is manipulated by operations such as `map` and `fold` (explained below) which process it as a whole and which happen to be implemented in parallel internally. These operations can be interleaved with sequential computations working on non-distributed data. In fact, the programmer views the computation as a *sequence* of parallel operations. Conceptually, this is almost as easy as sequential programming. Synchronization and communication problems like deadlocks and starvation cannot occur, since each skeleton

encapsulates the passing of low-level messages in a safe way. Currently, two main distributed data structures are offered by the library, namely:

```
template <class E> class DistributedArray{...}
template <class E> class DistributedMatrix{...}
```

where  $E$  is the type of the elements of the distributed data structure. Moreover, there are variants for sparse arrays and matrices, which we will not consider here. By instantiating the template parameter  $E$ , arbitrary element types can be generated. This shows one of the major features of distributed data structures and their operations in our framework. They are *polymorphic*. Moreover, a distributed data structure is split into several partitions, each of which is assigned to one processor participating in the data parallel computation. Currently, only block partitioning is supported. Future extensions by other partitioning schemes are planned.

Roughly, two classes of data parallel skeletons can be distinguished: computation skeletons and communication skeletons. *Computation skeletons* process the elements of a distributed data structure in parallel. Typical examples are the following methods in class `DistributedArray<E>`:

```
void mapIndexInPlace(E (*f)(int,E))
E fold(E (*f)(E,E))
```

`A.mapIndexInPlace(g)` applies a binary function  $g$  to each index position  $i$  and the corresponding array element  $A_i$  of a distributed array  $A$  and replaces  $A_i$  by  $g(i, A_i)$ . `A.fold(h)` combines all the elements of  $A$  successively by an associative binary function  $h$ . E.g. `A.fold(plus)` computes the sum of all elements of  $A$  (provided that  $E$  `plus(E,E)` adds two elements).

In order to prevent C++ experts from being annoyed, let us briefly point out that the types of the skeletons are in fact more general than shown above. E.g. the real type of `mapIndexInPlace` is:

```
template <class T> void mapIndexInPlace(T f)
```

Thus, the parameter  $f$  can not only be a C++ function of the mentioned type but also a so-called function object, i.e. an object representing a function of the corresponding type. In particular, such a function object can represent a partial application as we will explain below.

*Communication* consists of the exchange of the partitions of a distributed data structure between all processors participating in the data parallel computation. In order to avoid inefficiency, there is no implicit communication e.g. by accessing elements of remote partitions like in HPF [Ko94] or Pooma [Ka98], but the programmer has to control communication explicitly by using skeletons. Since there are no individual messages but only coordinated exchanges of partitions, deadlocks cannot occur. The most frequently used communication skeleton is

```
void permutePartition(int (*f)(int))
```

`A.permutePartition(f)` sends every partition  $A_{[j]}$  (located at processor  $j$ ) to processor  $f(j)$ .  $f$  needs to be bijective, which is checked at runtime. Some other communication skeletons correspond to MPI collective operations, e.g. `allToAll`, `broadcastPartition`, and `gather`. `A.broadcastPartition(j)`, for instance, replaces every partition of  $A$  by the one found at processor  $j$ .

Moreover, there are operations which allow one to access attributes of the local partition of a distributed data structure, e.g. `get`, `getLocalGlobal`, and `isLocal` (see Fig. 1) fetch an element of the local partition (using global or local indexing explained below, or combinations of both) and check whether a considered element is locally available, respectively. These operations are not skeletons but frequently used when implementing an argument function of a skeleton.

At first, skeletons such as `fold` and `scan` might seem equivalent to the corresponding MPI collective operations `MPI.Reduce` and `MPI.Scan`. However, they are more powerful due to the fact that the argument functions of all skeletons can be *partial applications* rather than just C++ functions. A skeleton essentially defines some parallel algorithmic structure, where the details can be fixed by appropriate argument functions. With partial applications as argument functions, these details can depend themselves on parameters, which are computed at runtime. For instance, in

```
template <class E> E plus3(E x, E y, E z){return x+y+z;}
...
int x = ... some computation ...;
A.mapIndexInPlace(curry(plus3)(x));
```

each element  $A_i$  of the distributed array  $A$  is replaced by `plus3(x,i,Ai)`. The “magic” `curry` function has been taken from the C++ template library `Fact` [St00], which offers functional-programming capabilities in C++ similar to FC++ [Sm04,MS00]. `curry` transforms an ordinary C++ function (such as `plus3`) into a function (object) which can be partially applied. Here, the computed value of  $x$  is given directly as a parameter to `plus3`. This results in a binary function, and this is exactly, what the `mapIndexInPlace` skeleton needs as parameter. Thus, `mapIndexInPlace` will supply the missing two arguments. Hence, partial applications as arguments provide the application-specific details to a generic skeleton and turn it into a specific parallel algorithm.

The code fragment in Fig. 1 taken from [Ku03] shows how skeletons and partial applications can be used in a more interesting example. It is a parallel implementation of (simplified) Gaussian elimination (ignoring potential division by 0 problems and numerical aspects). For the moment, this example shall only give you a flavor of the approach. We suggest that you return to the example after having read Subsections 2.2 and 2.3, which explain the skeletons in detail. Then, you will be able to completely understand the code.

The core is the function `gauss` in lines 11-16. In line 12 a  $p \times (n+1)$  distributed matrix `Pivot` is constructed and initialized with 0.0. According to the two last arguments of the constructor, it is split into  $p \times 1$  partitions, i.e. each of the  $p$  processors gets exactly one row. The  $n \times (n+1)$  coefficient matrix  $A$  is assumed

```

1 double copyPivot(const DistributedMatrix<double>& A,
2                 int k, int i, int j, double Aij){
3   return A.isLocal(k,k) ? A.get(k,j)/A.get(k,k) : 0.0;}

4 void pivotOp(const DistributedMatrix<double>& Pivot,
5             int rows, int firstrow, int k, double** LA){
6   for (int l=0; l<rows; l++){
7     double Alk = LA[l][k];
8     for (int j=k; j<=ProblemSize; j++)
9       if (firstrow+l == k) LA[l][j] = Pivot.getLocalGlobal(0,j);
10    else LA[l][j] -= Alk * Pivot.getLocalGlobal(0,j);}}

11 void gauss(DistributedMatrix<double>& A){
12   DistributedMatrix<double> Pivot(p,n+1,0.0,p,1);
13   for (int k=0; k<ProblemSize; k++){
14     Pivot.mapIndexInPlace(curry(copyPivot)(A)(k));
15     Pivot.broadcastRow(k);
16     A.mapPartitionInPlace(curry(pivotOp)(Pivot,n/p,A.getFirstRow(),k));}}

```

**Fig. 1.** Gaussian elimination with skeletons.

to be split into  $p \times 1$  partitions consisting of  $n/p$  consecutive rows. For instance, it could have been created by using the constructor:

```
DistributedMatrix<double> A(n,n+1,& init,p,1);
```

where the function `init` tells how to initialize each element of `A`. In each iteration of the loop in lines 13-16, the pivot row is divided by the pivot element and copied to a corresponding row of the matrix `Pivot` (line 14). In fact, the `mapIndexInPlace` skeleton tries to do this on every processor, but only at the processor owning the row it will actually happen. `copyPivot` sets all other elements to 0.0. The pivot row is then broadcast to every other row of matrix `Pivot` (line 15). Thus, every processor now has the pivot row. This row is then used by the corresponding processor to perform the pivot operation at every locally available element of matrix `A` (line 16). This is done by applying the `mapPartitionInPlace` skeleton to the pivot operation. More precisely, it is applied to a partial application of the C++ function `pivotOp` to the `Pivot` matrix and to three other arguments, which are a bit technical, namely to the number  $n/p$  of rows of `A` each processor has, to the index of the first row of the local partition of `A`, and to the index  $k$  of the pivot column (and row). The remaining fourth argument of `pivotOp`, namely the local partition of `A`, will be supplied by the `mapPartitionInPlace` skeleton. When applied to the mentioned arguments and to a partition of `A`, `pivotOp` (lines 4-10) performs the well-known pivot operation:

$$A_{l,j} = \begin{cases} A_{k,j}/A_{k,k}, & \text{if local row } l \text{ is the pivot row} \\ A_{l,j} - A_{l,j} \cdot A_{k,j}/A_{k,k}, & \text{otherwise} \end{cases}$$

for all elements  $A_{l,j}$  of the local partition. Note that `pivotOp` does not compute  $A_{k,j}/A_{k,k}$  itself, but fetches it from the  $j$ -th element of the locally available row

of `Pivot`. Since all rows of matrix `Pivot` are identical after the broadcast, the index of the row does not matter and we may take the first locally available row (in fact the only one), namely the one with local index 0. Thus, we find  $A_{k,j}/A_{k,k}$  in `Pivot.getLocalGlobal(0,j)`. Note that 0 is a local index referring to the local partition of `Pivot`, while  $j$  is a global one referring to matrix `Pivot` as a whole. Our skeleton library provides operations which can access array and matrix elements using local, global, and mixed indexing. The user may pick the most convenient one. Note that parallel programming based on message passing libraries such as MPI only provides local indexing.

It is also worth mentioning that for the above example the “non-standard” map operation `mapPartitionInPlace` is clearly more efficient (but slightly less elegant) than more classic map operations. `mapPartitionInPlace` manipulates a whole partition at once rather than a single element. This allows one to ignore such elements of a partition which need no processing. In the example, these are the elements to the left of the pivot column. Using classic map operations, one would have to apply the identity function to these elements, which causes substantial overhead.

## 2.2 Data Parallel Skeletons for Distributed Arrays

After this overview of data parallel skeletons, let us now consider them in more detail. This will set up the scene for the optimizations of sequences of these skeletons, which will be considered in Section 3. We will focus on skeletons for distributed arrays here. Skeletons for distributed matrices are very similar, and we will only sketch them in the next subsection. The following operations are needed to start and terminate a skeleton-based computation, respectively.

```
void InitSkeletons(int argc, char* argv[])
```

`InitSkeletons` needs to be called before the first skeleton is used. It initializes the global variables `sk_myid` and `sk_numprocs` (see below). The parameters `argc` and `argv` of `main` have to be passed on to `InitSkeletons`.

```
void TerminateSkeletons()
```

`TerminateSkeletons` needs to be called after the last skeleton has been applied. It terminates the skeleton computation cleanly.

The following global variables can be used by skeleton-based computations:

```
int sk_myid
```

This variable contains the number of the processor on which the considered computation takes place. It is often used in argument functions of skeletons, i.e. in functions telling how a skeleton should behave on the locally available data.

```
int sk_numprocs
```

contains the number of available processors.

All skeletons for distributed arrays (DAs) are methods of the corresponding class

```
template <class E> class DistributedArray {...}
```

It offers the public methods shown below. For every method which has a C++ function as an argument, there is an additional variant of it which may instead use a partial application with a corresponding type. As mentioned above, a partial application is generated by applying the function `curry` to a C++ function or by applying an existing partial application to additional arguments. For instance, in addition to

```
void mapInPlace(E (*f)(E))
```

(explained below) there is some variant of type

```
template <class F>
void mapInPlace(const Fct1<E,E,F>& f)
```

which can take a partial application rather than a C++ function as argument. Thus, if the C++ functions `succ` (successor function) and `add` (addition) have been previously defined, all elements of a distributed array `A` can be incremented by one either by using the first variant of `mapInPlace`

```
A.mapInPlace(succ)
```

or by the second variant

```
A.mapInPlace(curry(add)(1))
```

Of course, the second is more flexible, since the arguments of a partial application are computed at runtime. `Fct1<A,R,F>` is the type of a partial application representing a unary function with argument type `A` and result type `R`. The third template parameter `F` determines the computation rule needed to evaluate the application of the partial application to the missing arguments (see [KS02] for details). The type `Fct1<A,R,F>` is of internal use only. The user does not need to care about it. She only has to remember that skeletons may have partial applications instead of C++ functions as arguments and that these partial applications are created by applying the special, predefined function `curry` to a C++ function and by applying the result successively to additional arguments. Of course, there are also types for partial applications representing functions of arbitrary arity. In general, a partial application of type `Fcti<A1, ..., An, R, F>` represents a  $n$ -ary function with argument types  $A_1, \dots, A_n$  and result type `R`, for  $i \in \mathbb{N}$ .

Some of the following methods depend on some preconditions. An exception is thrown, if they are not met.  $p$  denotes the number of processors collaborating in the data parallel computation on the considered distributed array.

## Constructors

```
DistributedArray(int size, E (*f)(int))
```

creates a distributed array (DA) with `size` elements. The  $i$ -th element is initialized with `f(i)` for  $i = 0, \dots, \text{size} - 1$ . The DA is partitioned into



equally sized blocks, each of which is given to one of the  $p$  processors. More precisely, the  $j$ -th participating processor gets the elements with indices  $j \cdot \text{size}/p, \dots, (j+1) \cdot \text{size}/p - 1$ , where  $j = 0, \dots, p-1$ . We assume that  $p$  divides  $\text{size}$ .

**DistributedArray(int size, E initial)**

This constructor works as the previous. However, every array element is initialized with the same value `initial`. There are more constructors, which are omitted here in order to save space.

**Operations for Accessing Attributes of a Distributed Array.** The following operations access properties of the local partition of a distributed array. They are not skeletons themselves, but are frequently used when implementing argument functions of skeletons.

**int getFirst()**

delivers the index of the first locally available element of the DA.

**int getSize()**

returns the `size` (i.e. total number of elements) of the DA.

**int getLocalSize()**

returns the number of locally available elements.

**bool isLocal(int i)**

tells whether the  $i$ -th element is locally available.

**void setLocal(int i, E v)**

sets the value of the  $i$ -th locally available element to `v`. Note that the index  $i$  is referring to the local partition and not to the DA as a whole.

Precondition:  $0 \leq i \leq \text{getLocalSize}() - 1$ .

**void set(int i, E v)**

sets the value of the  $i$ -th element to `v`.

Precondition:  $j \cdot \text{size}/p \leq i < (j+1) \cdot \text{size}/p$ , if the local partition is the  $j$ -th partition of the DA ( $0 \leq j < p$ ).

**E getLocal(int i)**

returns the value of the  $i$ -th locally available element.

Precondition:  $0 \leq i \leq \text{getLocalSize}() - 1$ .

**E get(int i)**

delivers the value of the  $i$ -th element of the DA.

Precondition:  $j \cdot \text{size}/p \leq i < (j+1) \cdot \text{size}/p$ , where the local partition is the  $j$ -th partition of the DA ( $0 \leq j < p$ ).

Obviously, there are auxiliary functions which access a distributed array via a global index and others which access it via an index relative to the start of the local partition. Using global indexing is usually more elegant, while local indexing can be sometimes more efficient. MPI-based programs use local indexing only. Our skeleton library offers both, and it even allows to mix them as in the Gaussian elimination example (Fig. 1, lines 9 and 10). All these methods and most skeletons are inlined. Thus, there is no overhead for function calls when using them.

**Map Operations.** The `map` function is the most well-known and most frequently used higher-order function in functional languages. In its original formulation, it is used to apply an unary argument function to every element of a list and to construct a new list from the results. In the context of arrays and imperative or object oriented programming, this original formulation is of limited use. The computation related to an array element  $A_i$  often not only depends on  $A_i$  itself but also on its index  $i$ . In contrast to functional languages, imperative and object oriented languages allow also to update the element in place. Thus, there is no need to construct a new array, and time and space can be saved. Consequently, our library offers several variants of `map`, namely with and without considering the index and with and without update in place. In fact, `mapIndexInPlace` turned out to be the most useful one in many example applications. Some of the following variants of `map` require a template parameter `R`, which is the element type of the resulting distributed array.

```
template <class R> DistributedArray<R> map(R (*f)(E))
    returns a new DA, where element  $i$  has value  $f(\text{get}(i))$ 
    ( $0 \leq i \leq \text{getSize}() - 1$ ). It is partitioned as the considered DA.
template <class R> DistributedArray<R> mapIndex(R (*f)(int,E))
    returns a new DA, where element  $i$  has value  $f(i, \text{get}(i))$ 
    ( $0 \leq i \leq \text{getSize}() - 1$ ). It is partitioned as the considered DA.
void mapInPlace(E (*f)(E))
    replaces the value of each DA element with index  $i$  by  $f(\text{get}(i))$ ,
    where  $0 \leq i \leq \text{getSize}() - 1$ .
void mapIndexInPlace(E (*f)(int,E))
    replaces the value of each DA element with index  $i$  by  $f(i, \text{get}(i))$ ,
    where  $0 \leq i \leq \text{getSize}() - 1$ .
void mapPartitionInPlace(void (*f)(E*))
    replaces each partition  $P$  of the DA by  $f(P)$ .
```

**Zip Operations.** `zip` works on two equally partitioned distributed arrays of the same size and combines elements at corresponding positions. Just as for `map`, there are variants taking the index into account and variants updating one of the arrays in place. In the following `<class E2>` and `<class R>` are additional template parameters of the corresponding method.

```
DistributedArray<R> zipWith(const DistributedArray<E2>& b,
                           R (*f)(E,E2))
    returns a new DA, where element  $i$  has value  $f(\text{get}(i), b.\text{get}(i))$  and
     $0 \leq i \leq \text{getSize}() - 1$ . It is partitioned as the considered DA.
void zipWithInPlace(DistributedArray<E2>& b, E (*f)(E,E2))
    replaces the value of each DA element with index  $i$  by  $f(\text{get}(i), b.\text{get}(i))$ ,
    where  $0 \leq i \leq \text{getSize}() - 1$ .
DistributedArray<R> zipWithIndex(const DistributedArray<E2>& b,
                                 R (*f)(int,E,E2))
    returns a new DA, where element  $i$  has value  $f(i, \text{get}(i), b.\text{get}(i))$  and
     $0 \leq i \leq \text{getSize}() - 1$ . It is partitioned as the considered DA.
```

```
void zipWithIndexInPlace(const DistributedArray<E2>& b,
                        E (*f)(int,E,E2))
    replaces the value of each DA element with index  $i$  by  $f(i, \text{get}(i), b.\text{get}(i))$ ,
    where  $0 \leq i \leq \text{getSize}() - 1$ .
```

Both, `map` and `zipWith` (and their variants) require no communication.

**Fold and Scan.** `fold` (also known as reduction) and `scan` (often called parallel prefix) are computation skeletons, which require some communication internally. They combine the elements of a distributed array by an associative binary function.

```
E fold(E (*f)(E,E))
    combines all elements of the DA by the associative binary operation  $f$ , i.e.
    it computes  $f(\text{get}(0), f(\text{get}(1), \dots f(\text{get}(n-2), \text{get}(n-1)) \dots))$ 
    where  $n = \text{getSize}()$ . Precondition:  $f$  is associative (this is not checked).
void scan(E (*f)(E,E))
    replaces every element with index  $i$  by
     $f(\text{get}(0), f(\text{get}(1), \dots f(\text{get}(i-1), \text{get}(i)) \dots))$ ,
    where  $0 \leq i \leq \text{getSize}() - 1$ .
    Precondition:  $f$  is associative (this is not checked).
```

Both, `fold` and `scan`, require  $\Theta(\log p)$  messages (of size  $\text{size}(E)$  and  $n/p \cdot \text{size}(E)$ , respectively) per processor [BK98].

**Communication Operations.** As mentioned already, communication skeletons rearrange the partitions of a distributed data structure. This is done in such a way that the index range assigned to each processor is not changed. Only the corresponding values are replaced by those from another partition.

```
void permutePartition(int (*f)(int))
    replaces every partition  $f(i)$  by partition  $i$  where  $0 \leq i \leq p - 1$ .
    Precondition:  $f : \{0, \dots, p - 1\} \rightarrow \{0, \dots, p - 1\}$  is bijective. This is checked
    at runtime.
void permute(int (*f)(int))
    replaces every element with index  $f(i)$  by the element with index  $i$ 
    where  $0 \leq i \leq \text{getSize}() - 1$ .
    Precondition:  $f : \{0, \dots, \text{getSize}() - 1\} \rightarrow \{0, \dots, \text{getSize}() - 1\}$  is bijective.
    This is checked at runtime.
void broadcastPartition(int i)
    replaces every partition of the DA by partition  $i$ .
    Precondition:  $0 \leq i \leq p - 1$ .
void broadcast(int index)
    replaces every element of the DA by the element with index  $i$ .
    Precondition:  $0 \leq i \leq \text{getSize}() - 1$ .
```

```
void allToAll(const DistributedArray<int *>& Index, E dummy)
```

uses parts of the local partition to replace parts of each other partition. **Index** is a DA of type  $\text{int}[p + 1]$  indicating which part goes where. Depending on **Index**, some processors may get more elements than others. The unused elements of each partition are filled with **dummy**.

Another communication skeleton, **gather**, is a bit special, since it does not rearrange a distributed array, but it copies it to an ordinary (non-distributed) array. Since all non-distributed data structures and the corresponding computations are replicated on each processor participating in a data parallel computation [BK98], **gather** corresponds to the MPI operation **MPI\_Allgather** rather than to **MPI\_Gather**.

```
void gather(E b[])
```

copies all elements of the considered DA to the non-distributed array **b**.

**permute** and **permutePartition** require a single send and receive per processor, while **broadcast**, **broadcastPartition**, and **gather** need  $\Theta(\log p)$  communication steps (with message sizes  $\text{size}(E)$  and  $n/p \cdot \text{size}(E)$ , respectively). **allToAll** requires  $\Theta(p)$  communication steps.

## Other Operations

```
DistributedArray<E> copy()
```

generates a copy of the considered DA.

```
void show()
```

prints the DA on standard output.

## 2.3 Data Parallel Skeletons for Distributed Matrices

The operations for distributed matrices (DM) are similar to those for distributed arrays. Thus, we will sketch them only briefly here. The constructors specify the number of rows and columns, tell how the elements shall be initialized, and give the number of partitions in vertical and horizontal direction. E.g. the constructor

```
DistributedMatrix(int n, int m, E x0, int v, int h)
```

constructs a distributed  $n \times m$  matrix, where every element is initialized with **x0**. The matrix is split into  $v \times h$  partitions. As for DAs, there are other constructors which compute the value of each element depending on its position using a C++ function or function object.

The operations for accessing attributes of a DM now have to take both dimensions into account. Consequently, there are operations delivering the number of (local or global) rows and columns, the index of the first locally available row and column, and so on. As shown in the Gaussian-elimination example, the operations for accessing an element of a DM can use any combination of local and global indexing. For instance, **M.get(*i*,*j*)** fetches (global) element  $M_{i,j}$  of DM

M, while `M.getLocalGlobal( $l, j$ )` accesses the element in local row  $l$  and global column  $j$ .

There are also straightforward variants of map, zip, fold, and scan operations as well as communication operations such as permutation, broadcast, and all-to-all, e.g.

```
void permutePartition(int (*f)(int,int), int (*g)(int,int))
```

replaces every partition  $P_{i',j'}$  by partition  $P_{i,j}$ , where  $i' = f(i, j)$  and  $j' = g(i, j)$ . It is checked at runtime whether the resulting mapping is bijective.

Additionally, there are rotation operations, which allow to rotate the partitions of a DM cyclically in vertical or horizontal direction. In fact, these are special cases of `permutePartition`, e.g.

```
void rotateRows(int (*f)(int))
```

cyclically rotates the partitions in row  $i$  by  $f(i)$  partitions to the right. Negative values for  $f(i)$  correspond to rotations to the left.

### 3 Optimization of Sequences of Skeletons

As explained above, the user of the data parallel skeletons has a global view of the parallel computation. An advantage of this global view is that it simplifies global optimizations compared to the local view of MPI-based processes. In particular it enables algebraic transformations of sequences of skeletons, much in the spirit of the well-known Bird-Meertens formalism (see e.g. [Bi88,Bi89,GL97]). For instance

```
A.mapIndexInPlace(f);
A.mapIndexInPlace(g);
```

can be replaced by

```
A.mapIndexInPlace(curry(compose)(g)(f));
```

where `compose` is assumed to be a C++ implementation of a composition function, for instance by:

```
template <class C1, class C2, class C3>
inline C3 compose(C3 (*f)(C2), C2 (*g)(C1), C1 x){return f(g(x));}
```

In the sequel, we will discuss a few such transformations, however without attempting to generate a complete set of them. The idea is to consider a few optimizations and check out their impact based on some experimental results for a few small benchmarks. The final aim will be to create an automatic optimizer which detects sequences of skeletons and replaces them by some semantically equivalent but more efficient alternative sequences of skeletons, as in the above example. However, this aim is far beyond the scope of this chapter. The mentioned optimizer will need a cost model of each skeleton in order to estimate the

**Table 1.** Runtimes on 4 processors for sequences of skeletons working on a distributed array of  $n$  elements and corresponding speedups.

example	$n$	original	combined	speedup
<code>permutePartition</code> $\circ$ <code>permutePartition</code>	$2^7$	37.00 $\mu$ s	16.43 $\mu$ s	2.25
<code>mapIndexInPlace</code> $\circ$ <code>mapIndexInPlace</code>	$2^7$	0.69 $\mu$ s	0.40 $\mu$ s	1.73
<code>mapIndexInPlace</code> $\circ$ <code>fold</code>	$2^7$	144.82 $\mu$ s	144.60 $\mu$ s	1.00
<code>mapIndexInPlace</code> $\circ$ <code>scan</code>	$2^7$	144.52 $\mu$ s	144.48 $\mu$ s	1.00
<code>multiMapIndexInPlace</code>	$2^7$	0.67 $\mu$ s	0.65 $\mu$ s	1.03
<code>mapIndexInPlace</code> $\circ$ <code>permutePartition</code>	$2^7$	16.60 $\mu$ s	31.43 $\mu$ s	0.53
<code>permutePartition</code> $\circ$ <code>permutePartition</code>	$2^{19}$	10.76 ms	5.32 ms	2.02
<code>mapIndexInPlace</code> $\circ$ <code>mapIndexInPlace</code>	$2^{19}$	2.79 ms	1.59 ms	1.75
<code>mapIndexInPlace</code> $\circ$ <code>fold</code>	$2^{19}$	2.97 ms	2.67 ms	1.11
<code>mapIndexInPlace</code> $\circ$ <code>scan</code>	$2^{19}$	6.01 ms	5.83 ms	1.03
<code>multiMapIndexInPlace</code>	$2^{19}$	2.77 ms	2.62 ms	1.06
<code>mapIndexInPlace</code> $\circ$ <code>permutePartition</code>	$2^{19}$	6.68 ms	5.72 ms	1.17

**Table 2.** Speedups for the combination of two rotations of a  $n \times n$  matrix depending on the number  $p$  of processors.

$n \setminus p$	4	8	16
16	1.64	1.91	1.89
128	1.71	1.92	1.70
1024	1.48	1.81	1.72

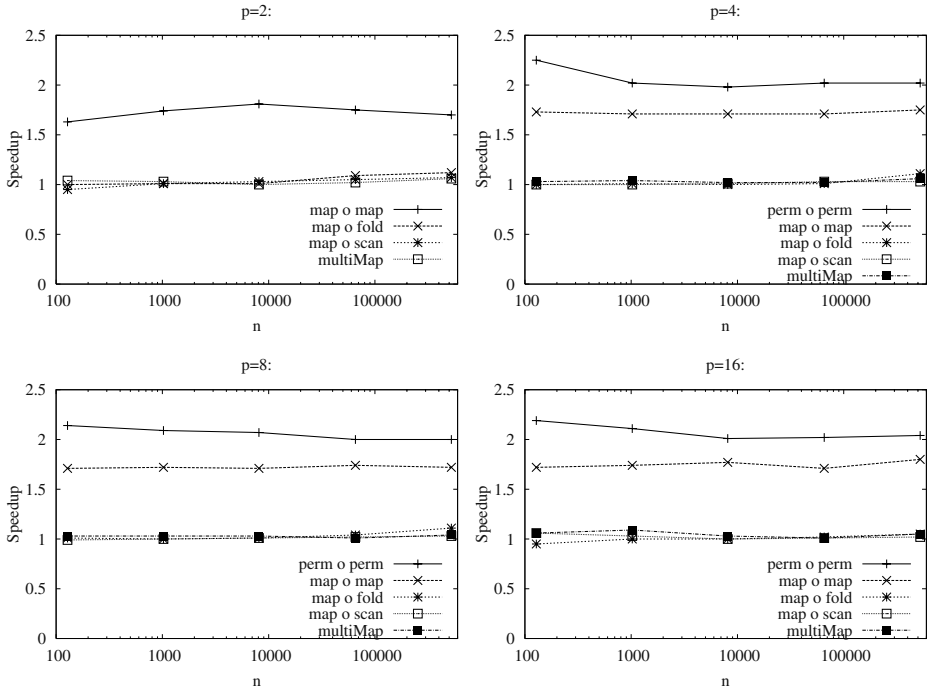
runtimes of alternative sequences of skeletons. A corresponding cost model for most of the mentioned skeletons can be found in [BK98]. Other cost models for skeletons and similar approaches like the bulk synchronous parallel (BSP) model [SH97] are presented in e.g. [MC95,SD98]. A BSP computation is a sequence of so-called supersteps each consisting of a computation and a global communication phase. This restricted model facilitates the estimation of costs. Our data parallel skeletons are similarly easy to handle.

The transformation in the above example is an instance of a more general pattern, where a sequence of the same skeleton is replaced by a single occurrence of this skeleton. This pattern can also be applied to e.g. other variants of `map`, `permutePartition`, `rotateRows`, and `rotateCols`.

We have compared the runtimes of the original and the transformed sequence of skeleton calls on the IBM cluster at the University of Münster. This machine [ZC03] has 94 compute nodes, each with a 2.4 GHz Intel Xeon processor, 512 KB L2 Cache, and 1 GB memory. The nodes are connected by a Myrinet and running RedHat Linux 7.3 and the MPICH-gm implementation of MPI.

Fig. 2 and (partly) Table 1 show the speedups which could be achieved for different array sizes and numbers of processors. It is interesting to note that the problem size and the number of processors have little influence on the speedup.

By composing two permutation skeletons, the expected speedup of about 2 could be obtained. Combining two rotations of a distributed matrix leads to speedups between 1.5 and 1.9 (see Table 2). There are more opportunities for



**Fig. 2.** Speedups for the combination of skeletons on 2, 4, 8, and 16 processors. On 2 processors, the combination of permutations does not make sense. “map” stands for `mapIndexInPlace` and “perm” for `permutePartition`.

combining sequences of communication skeletons as shown in Table 3. Transformations involving rotations are shown for distributed matrices rather than arrays. Since these transformations can be rarely applied in practice (as explained below), we do not consider them further here. Moreover, we omit the corresponding correctness proofs, since the transformations are rather straightforward.

Combining two computation skeletons such as `map` corresponds to *loop fusion* (see e.g. [ZC90]). For `mapIndexInPlace` this roughly means that internally

```
for(int i = 0; i<localsize; i++)
    a[i] = f(i+first,a[i]);
for(int i = 0; i<localsize; i++)
    a[i] = g(i+first,a[i]);
```

is replaced by

```
for(int i = 0; i<localsize; i++)
    a[i] = g(i+first,f(i+first,a[i]));
```

Obviously, one assignment and the overhead for the control of one loop are saved. It is well-known that loop fusion is only possible if the considered loops do not

**Table 3.** Transformations for sequences of communication skeletons working on a distributed array **A** and a distributed matrix **M**, respectively.  $h'(x) = x + h(x) \bmod m$ , where  $m$  is the number of partitions in each row (or column). **f** and **g** are analogously transformed to **f'** and **g'**, respectively.

original	combined
A.permute(f); A.permute(g);	A.permute(curry(compose)(g)(f)); (analogously for permutePartition)
A.permute(f); A.broadcast(k);	A.broadcast( $f^{-1}(k)$ ); (analogously for permutePartition)
A.broadcast(k); A.permute(f);	A.broadcast(k); (analogously for permutePartition)
A.broadcast(k); A.broadcast(i);	A.broadcast(k);
M.permutePartition(f,g); M.rotateRows(h);	M.permutePartition(curry(compose)(h')(f),g);
M.rotateRows(h); M.permutePartition(f,g);	M.permutePartition(curry(compose)(f)(h'),g);
M.permutePartition(f,g); M.rotateCols(h);	M.permutePartition(f,curry(compose)(h')(g));
M.rotateCols(h); M.permutePartition(f,g);	M.permutePartition(f,curry(compose)(g)(h'));
M.broadcast(i,j); M.rotateRows(h);	M.broadcast(i,j); (analogously for rotateCols)
M.rotateRows(h); M.broadcast(i,j);	M.broadcast( $h'^{-1}(i),j$ ); (analogously for rotateCols)
M.rotateRows(f); M.rotateRows(g);	M.rotateRows(curry(compose)(g)(f)); (analogously for rotateCols)
M.rotateRows(f); M.rotateCols(g);	M.permutePartition(f',g');
M.rotateCols(f); M.rotateRows(g);	M.permutePartition(g',f');

depend on each other. When combining two calls of `mapIndexInPlace`, this is typically the case. Note however that **f** and **g** may be partial applications having array **a** as parameter. In this case, loop fusion cannot (easily) be applied. Moreover, loop fusion is not always a good idea. For instance, in some cases the caching behavior may deteriorate to an extent that the saved overhead for loop control is outweighed. In our experiments (see Fig. 2), we could achieve a speedup of about 1.7, when composing two occurrences of `map` (in fact `mapIndexInPlace`). However, this is only possible for simple argument functions. If each call to the argument function causes a complex computation, the advantage of loop fusion almost disappears. If we choose an argument function, where each call causes a loop of 1000 iterations to be executed (instead of one iteration as in Fig. 2), the speedup decreases to about 1.05 (see Table 4).

A similar kind of loop fusion can also be performed when combining different skeletons, e.g. `mapIndexInPlace` and `fold` or `scan` (see Fig. 2). However, this



**Table 4.** Runtimes for the combination of two occurrences of `mapIndexInPlace` working on a distributed array with  $n$  elements on 4 processors depending on the number  $k$  of iterations executed for each call to the argument function of `mapIndexInPlace`.

$k$	$n$	original	combined	speedup
1	$2^7$	0.69 $\mu$ s	0.40 $\mu$ s	1.73
1000	$2^7$	21.23 $\mu$ s	19.79 $\mu$ s	1.07
1	$2^{19}$	2.79 ms	1.59 ms	1.75
1000	$2^{19}$	83.00 ms	80.12 ms	1.04

requires not only a transformation of the application program, but the skeleton library also has to be extended by the required combined skeletons. For instance,

```
A.mapIndexInPlace(f);
result = A.fold(g);
```

is replaced by

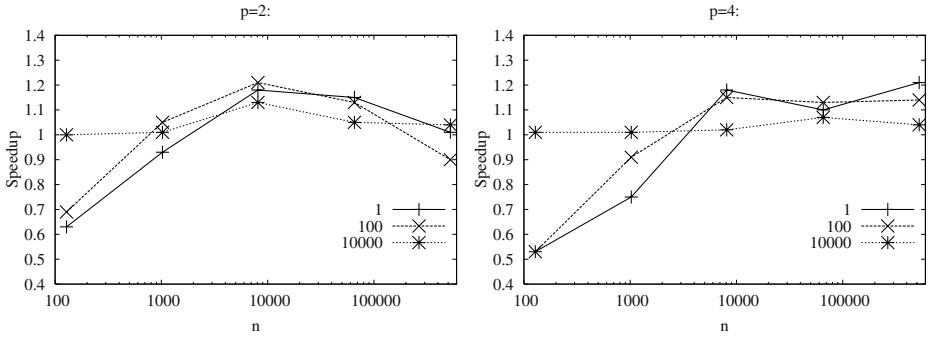
```
result = A.mapIndexInPlaceFold(f,g);
```

in the application program, and the new skeleton `mapIndexInPlaceFold` with obvious meaning is added to the library. Combining `map` and `scan` is done analogously. As Fig. 2 shows, speedups of up to 1.1 can be obtained for big arrays, and a bit less for smaller arrays. Again, a simple argument function was used here. With more complex argument functions, the speedup decreases. The same approach can be applied for fusing:

- two calls to `zipWith` (and variants),
- `zipWith` and `map`, `fold`, or `scan`,
- a communication skeleton (such as `permute`, `broadcast` and `rotate`) and a computation skeleton (such as `map`, `fold`, and `scan`) (and vice versa).

Let us consider the latter in more detail. If implemented in a clever way, it may (in addition to loop fusion) allow one to *overlap communication and computation* and hence provide a new source for improvements. For the combination of `mapIndexInPlace` and `permutePartition`, we observed a speedup of up to 1.25 (see Fig. 3). Here, the speedup heavily depends on a good balance between computation and communication. If the arrays are too small and the amount of computation performed by the argument function of `mapIndexInPlace` is not sufficient, the overhead of sending more (but smaller) messages does not pay off, and a slowdown may occur. Also if the amount of computation exceeds that of communication, the speedup decreases again. With an optimal balance between computation and communication (and a more sophisticated implementation of the `mapIndexInPlaceFold` skeleton) speedups up to 2 are theoretically possible.

We have investigated a few example applications (see [Ku03]) such as matrix multiplication, FFT, Gaussian elimination, all pairs shortest paths, samplesort, TSP (traveling salesperson problem), and bitonic sort in order to check which of the optimizations discussed above can be applied more or less frequently in



**Fig. 3.** Speedups for the combination of `mapIndexInPlace` and `permutePartition` on 2 and 4 processors depending on the number of iterations (1-10000) of a simple loop that each call to the argument function requires.

practice. Unfortunately, there was no opportunity to combine communication skeletons. This is not too surprising. If such a combination is possible, it is typically performed by the programmer directly, and there is no need for an additional optimization. However automatically generated code as well as programs written by less skilled programmers might still benefit from such transformations. Moreover, we did find opportunities for combining map and fold, e.g. in the TSP example [KC02], and also for combining two occurrences of map. In our examples, there was no instance of a combination of map and scan. However, you can find one in [Go04]. Very often, there were sequences consisting of a communication skeleton followed by a computation skeleton or vice versa as in our `mapIndexInPlace``permutePartition` example (see Fig. 2). This is not surprising, since this alternation of communication and computation is what the BSP approach [SH97] is all about; and using data parallel skeletons is somehow similar to BSP.

In several examples, we also found sequences of maps, which were manipulating different arrays. Here, the above transformation could not be applied. However, since we noted the importance of this pattern in practice, we have added a new skeleton `multiMap` (with the usual variants such as `multiMapIndexInPlace`) to our library. `multiMap` combines two maps working on two equally partitioned distributed arrays (or matrices) of the same size. Internally, this results in a similar loop fusion as for the combination of maps working on the same array. For instance

```
A.mapIndexInPlace(f);
B.mapIndexInPlace(g);
```

can be replaced by: `multiMapIndexInPlace(A,f,B,g);`

In our example code, this transformation caused a speedup of up to 1.08 (see Fig. 2). Since `multiMap` is not particularly nice, it is supposed to be used by an optimizer only, and not explained to the user.

## 4 Conclusions and Future Work

We have explained the data parallel skeletons of our C++ skeleton library in detail. Moreover, we have considered the optimization of sequences of these skeletons. When combining sequences of communication skeletons such as permutations, broadcasts, and rotations, we observed the expected speedup. Unfortunately, there are little opportunities in practical applications, where communication skeletons can be successfully fused. For computation skeletons such as map and fold and, in particular, for sequences of communication and computation skeletons, the situation is different. Practical applications often contain such sequences, such that the corresponding transformation can be applied. This essentially leads to some kind of loop fusion, and, in the case of combined communication and computation skeletons, to overlapping communication and computation. Speedups of a few percent could be achieved this way.

In the future, we plan to develop an automatic optimizer, which replaces sequences of skeletons by semantically equivalent but more efficient sequences. The purpose of the presented case study was to check out the potential of such an optimizer.

## Acknowledgements

I would like to thank the anonymous referees for lots of helpful suggestions.

## References

- [BG02] Bischof, H., Gorlatch, S.: Double-Scan: Introducing and Implementing a New Data-Parallel Skeleton. In Monien, B., Feldmann, R., eds.: Euro-Par'02. LNCS 2400, Springer-Verlag (2002) 640-647
- [Bi88] Bird, R.: Lectures on Constructive Functional Programming, In Broy, M., ed.: Constructive Methods in Computing Science, NATO ASI Series. Springer-Verlag (1988) 151-216
- [Bi89] Bird, R.: Algebraic identities for program calculation. *The Computer Journal* 32(2) (1989) 122-126
- [BK96] Botorog, G.H., Kuchen, H.: Efficient Parallel Programming with Algorithmic Skeletons. In Bougé, L. et al., eds.: Euro-Par'96. LNCS 1123, Springer-Verlag (1996) 718-731
- [BK98] Botorog, G.H., Kuchen, H.: Efficient High-Level Parallel Programming, *Theoretical Computer Science* 196 (1998) 71-107
- [Co89] Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989)
- [Da93] Darlington, J., Field, A.J., Harrison T.G., et al: Parallel Programming Using Skeleton Functions. PARLE'93. LNCS 694, Springer-Verlag (1993) 146-160
- [Da95] Darlington, J., Guo, Y., To, H.W., Yang, J.: Functional Skeletons for Parallel Coordination. In Hardidi, S., Magnusson, P.: Euro-Par'95. LNCS 966, Springer-Verlag (1995) 55-66
- [DP97] Danelutto, M., Pasqualetti, F., Pelagatti S.: Skeletons for Data Parallelism in p3l. In Lengauer, C., Griebel, M., Gorlatch, S.: Euro-Par'97. LNCS 1300, Springer-Verlag (1997) 619-628

- [FO92] Foster, I., Olson, R., Tuecke, S.: Productive Parallel Programming: The PCN Approach. *Scientific Programming* 1(1) (1992) 51-66
- [GH95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison-Wesley (1995)
- [GL99] Gropp, W., Lusk, E., Skjellum, A.: *Using MPI*. MIT Press (1999)
- [GL97] Gorlatch, S., Lengauer, C.: (De)Composition Rules for Parallel Scan and Reduction, 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97). *IEEE* (1997) 23-32
- [Go04] Gorlatch, S.: Optimizing Compositions of Components in Parallel and Distributed Programming (2004) In this volume
- [GW99] Gorlatch, S., Wedler, C., Lengauer, C.: Optimization Rules for Programming with Collective Operations, 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (1999) 492-499
- [Ka98] Karmesin, S., et al.: Array Design and Expression Evaluation in POOMA II. *ISCOPE'98* (1998) 231-238
- [KC02] Kuchen, H., Cole, M.: The Integration of Task and Data Parallel Skeletons. *Parallel Processing Letters* 12(2) (2002) 141-155
- [Ko94] Koelbl, C.H., et al.: *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press (1994)
- [KP94] Kuchen, H., Plasmeijer, R., Stoltze, H.: Efficient Distributed Memory Implementation of a Data Parallel Functional Language. *PARLE'94*. LNCS 817, Springer-Verlag (1994) 466-475
- [KS02] Kuchen, H., Striegnitz, J.: Higher-Order Functions and Partial Applications for a C++ Skeleton Library. *Joint ACM Java Grande & ISCOPE Conference*. ACM (2002) 122-130
- [Ku02] Kuchen, H.: A Skeleton Library. In Monien, B., Feldmann, R.: *Euro-Par'02*. LNCS 2400, Springer-Verlag (2002) 620-629
- [Ku03] Kuchen, H.: The Skeleton Library Web Pages.  
<http://danae.uni-muenster.de/lehre/kuchen/Skeletons/>
- [Le04] Lengauer, C.: Program Optimization in the Domain of High-Performance Parallelism. (2004) In this volume
- [MC95] W. McColl: Scalable Computing. In van Leuwen, J., ed.: *Computer Science Today*, LNCS 1000, Springer-Verlag (1995) 46-61
- [MS00] McNamara, B., Smaragdakis, Y.: Functional Programming in C++. *ICFP'00*. ACM (2000) 118-129
- [SD98] Skillicorn, D., Danelutto, M., Pelagatti, S., Zavanella, A.: Optimising Data-Parallel Programs Using the BSP Cost Model. In Pritchard, D., Reeve, J.: *Euro-Par'98*. LNCS 1470, Springer-Verlag (1998) 698-703
- [SH97] Skillicorn, D., Hill, J.M.D., McColl, W.: Questions and Answers about BSP. *Scientific Programming* 6(3) (1997) 249-274
- [Sk94] Skillicorn, D.: *Foundations of Parallel Programming*. Cambridge U. Press (1994)
- [Sm04] Smaragdakis, Y.: A Personal Outlook of Generator Research (2004) In this volume
- [St00] Striegnitz, J.: Making C++ Ready for Algorithmic Skeletons. Tech. Report IB-2000-08, <http://www.fz-juelich.de/zam/docs/autoren/striegnitz.html>
- [ZC90] Zima, H.P., Chapman, B.M.: *Supercompilers for Parallel and Vector Computers*. ACM Press/Addison-Wesley (1990)
- [ZC03] ZIV-Cluster: <http://zivcluster.uni-muenster.de/>

# Domain-Specific Optimizations of Composed Parallel Components

Sergei Gorlatch

University of Münster, Germany  
gorlatch@math.uni-muenster.de

**Abstract.** Generic components, customizable for specific application domains, are being increasingly used in the software development for parallel and distributed systems. While previous research focused mostly on single components, e.g. *skeletons*, and their properties, our work addresses the optimization opportunities for the typical situation that several calls to components are composed in a large application. We present an overview of the recently developed optimizations of parallel and distributed components at three abstraction levels: 1) fusing compositions of basic skeletons, expressed as collective operations in MPI (Message Passing Interface), 2) developing cost-optimal implementations for skeleton compositions using domain-specific data structures, and 3) optimizing remote execution of components' compositions in the context of Java RMI (Remote Method Invocation) in Grid-like distributed environments. We demonstrate our optimizations using application case studies and report experimental results on the achieved performance improvements.

## 1 Introduction

Various kinds of program components are being increasingly used in software development for modern parallel and distributed systems. One popular class of components in parallel programming are *skeletons* that are viewed formally as higher-order functions [1]. Skeletons capture typical algorithmic patterns, represent them in generic form to the user and provide possibilities for domain-specific customization. While being parameterized generic programming constructs, skeletons can be optimized for particular application domains by the user who provides suitable, domain-specific parameter values that can be either data or code.

The future of component-based, domain-specific program development will require making components first-class objects and creating theories and mechanisms to manipulate compositions of components. There has been active research on developing specific skeletons, studying their features and performance properties. While such research has yielded useful results, it does not address the whole range of relevant challenges, one of the most important being how to compose parallel components efficiently in one application.

We present here an overview of our recent results covering quite a wide range of program components used in the parallel and distributed setting. Our work can be viewed as a step on the way to developing a mathematical science of parallel components' composition.

The structure of the paper, with its main contributions, is as follows:

- Section 2 introduces the programming approach using skeletons, and suggests three abstraction levels at which domain-specific optimizations of skeleton compositions can be accomplished.
- In Section 3, we study basic parallel skeletons implemented by the collective operations of MPI, present two rules for fusing a composition of collective operations, and describe the use of the rules in the program design process.
- Section 4 demonstrates how a new, cost-optimal parallel implementation for a composition of two skeletons can be developed; we describe an application case study and report experimental results on parallel machines.
- In Section 4, we optimize compositions of remote method calls in the distributed setting based on Java RMI and illustrate this by a case study that includes performance measurements on a Grid-like system.

We conclude by discussing our results in the context of related work.

## 2 Levels and Criteria of Domain-Specific Optimizations

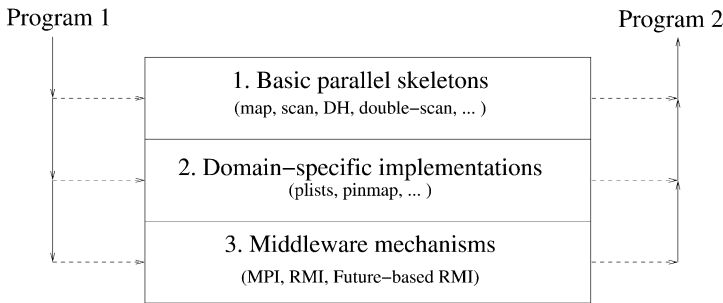
An important advantage of programming using components in general and skeletons in particular is that they come with generic, efficient implementations geared to different middleware mechanisms and different kinds of target machine architectures. This frees the user from the need to develop an implementation from scratch for each (part of a) new application: the application has to be expressed in terms of the available skeletons.

The key motivation for our work is that many applications cannot be expressed using a single component. Instead, a composition of several components, each customized in a specific manner, is used. For a particular application, the solution is composed of several components, each instantiated with concrete parameter values. We address the challenging, albeit so far largely neglected problem of optimizing a composition of two (or more) skeletons.

We will consider three abstraction levels at which optimization of particular compositions of skeletons in parallel programs can be accomplished (see Figure 1):

1. Basic parallel skeletons: In this case, a particular composition of two or several basic skeletons can be reformulated as a semantically equivalent composition of these and possibly other skeletons. Optimization is done as a semantics-preserving transformation applicable under particular domain-specific conditions, e.g. associativity of the involved operations.

2. Domain-specific implementations: Here we go one level deeper in the abstraction hierarchy and consider the case that a given composition cannot be optimized using only basic skeletons. New, domain-specific skeletons and data structures are to be introduced for the purpose of optimization.
3. Middleware mechanisms: This is the lowest abstraction level, where optimization addresses not skeletons themselves but rather their underlying execution mechanism, with the aim to speed-up the execution of a particular skeleton composition.



**Fig. 1.** Compositions of skeletons are optimized at three levels

At each level of optimization, it must be decided which goal is pursued thereby. In the domain of parallel programming for high-performance computers, two main criteria are used: runtime and/or cost:

- The reduced runtime of a program, measured in the number of elementary operations, is usually the ultimate goal when expensive computational and communicational resources are bundled in a parallel or distributed system. Program developers are interested in both asymptotic time estimates (runtime behaviour depending on the problem size and the number of processors) and also more practice-oriented approaches (where the goal is to predict the absolute runtime of a program as precisely as possible).
- An important efficiency criterion for parallel algorithms is their *cost*, which is defined as the product of the required time and the number of processors used, i. e.  $c = p \cdot t_p$  [2]. A parallel implementation is called *cost-optimal* on  $p$  processors, iff its cost equals the cost on one processor, i. e.  $p \cdot t_p \in \Theta(t_{seq})$ . Here and in the remainder of this paper, we use the classical notations  $o$ ,  $O$ ,  $\omega$  and  $\Theta$  to describe asymptotic behaviour [3].

The next three sections deal with compositions of parallel skeletons at the three abstraction levels shown in Figure 1.

### 3 Compositions of Basic Skeletons

In this section, our components of interest are basic parallel skeletons. On the one hand, they are widely used in many skeleton-based systems. On the other hand, these skeletons are implemented as collective operations, which are an important part of the MPI (Message Passing Interface) standard [4]. Compared with the more traditional individual communication statements like `Send` and `Receive`, collective operations have many advantages: parallel programs become better structured, less error-prone, and their performance can be better predicted [5].

#### 3.1 Modeling MPI Programs

Message-passing programs are usually written in the SPMD (Single Program Multiple Data) style, such that processor program is a sequence of computation and communication stages. We use a simplified pseudocode for representing such programs: computations are expressed by function calls, and communications by calls to the MPI collective operations. For example, the following program is a sequential composition of three stages: computation stage, `f`, followed by two stages, `scan` and `reduction`, that perform both communication and computation:

```

Program Example (my_pid, ...);
  f (...);
  MPI_Scan (⊗);
  MPI_Reduce (⊕);

```

(1)

To allow formal reasoning about message-passing programs, in particular about compositions in them, we represent programs in the classical functional framework. To keep the presentation simple, our MPI programs are defined over one-dimensional arrays, which are modeled by non-empty, finite lists.

The computation stage of (1), denoted by `f (...)`, means that every processor computes function `f` on its local data, independently of the other processors. This can be modeled by the functional *map*, which applies the unary function *f* to all elements of a list (function application is denoted by juxtaposition, has the highest binding power, and associates to the left):

$$\text{map } f [x_1, x_2, \dots, x_n] = [f x_1, f x_2, \dots, f x_n] \quad (2)$$

The collective operation `MPI_Reduce` in (1) combines data in the processors, with an associative binary operator, which can be either user-defined or predefined, e.g., addition or maximum. We capture this genericity by using parameter operation  $\oplus$  in `MPI_Reduce(⊕)`; other arguments of MPI calls are usually omitted in this paper for the sake of brevity. We model the reduction by the functional *red*, which expects an associative operator,  $\oplus$ , and a list:

$$\text{red}(\oplus) [x_1, x_2, \dots, x_n] = x_1 \oplus x_2 \oplus \dots \oplus x_n \quad (3)$$

Another important collective operation is `MPI_Scan(⊗)`, which performs the “prefix sums” calculation: on each processor, the result is an accumulation of the data from processors with smaller ranks, using the associative operator  $\otimes$ .



The corresponding functional version of scan is as follows:

$$\text{scan}(\otimes)[x_1, x_2, \dots, x_n] = [x_1, x_1 \otimes x_2, \dots, x_1 \otimes x_2 \otimes \dots \otimes x_n] \quad (4)$$

We call *map*, *red* and *scan* “skeletons” because each describes a whole class of functions, obtainable by substituting domain-specific operators for  $\oplus$  and  $f$ .

The typical structure of MPI programs is a sequential composition of computations and collective operations that are executed one after another. We model this by functional composition  $\circ$ :

$$(f \circ g)x \stackrel{\text{def}}{=} f(gx) \quad (5)$$

To summarize, the MPI program (1) is expressed functionally as follows:

$$\text{Example} = \text{red}(\oplus) \circ \text{scan}(\otimes) \circ \text{map } f \quad (6)$$

Of course, modeling imperative programs functionally requires that side effects, aliasing, etc. are taken into account. Corresponding techniques known from the literature on modern compilers [6] are not considered here. Note also that the composed functions in (6) are computed from right to left: *map*, then *scan*, etc.

### 3.2 Composition Rules for Collective Operations

Complex parallel programs are usually “engineered” by composing several parallel skeletons. Sequential composition, expressed by  $;$  in MPI and by  $\circ$  functionally, plays a key role in program construction. The compositions of collective operations may be quite expensive, especially if many processors of a machine have to synchronize their activity at the points of composition. We present two exemplary optimization rules, proved in prior work [7], that exploit specific domain-specific properties of the parameter operators.

Our first rule “fuses” a scan followed by a reduction into one reduction:

**Theorem 1 (Scan-Reduce Composition).** *For arbitrary binary associative operators  $\oplus$  and  $\otimes$ , if  $\otimes$  distributes over  $\oplus$ , then*

$$\text{red}(\oplus) \circ \text{scan}(\otimes) = \pi_1 \circ \text{red}(\langle \oplus, \otimes \rangle) \circ \text{map pair} \quad (7)$$

The functions used in (7) are defined as follows:

$$\text{pair } a \stackrel{\text{def}}{=} (a, a), \quad (8)$$

$$\pi_1(a, b) \stackrel{\text{def}}{=} a \quad (9)$$

$$(s_1, r_1) \langle \oplus, \otimes \rangle (s_2, r_2) \stackrel{\text{def}}{=} (s_1 \oplus (r_1 \otimes s_2), r_1 \otimes r_2) \quad (10)$$

For MPI programs, Theorem 1 provides the following transformation:

$$\left[ \begin{array}{l} \text{MPI\_Scan}(\otimes); \\ \text{MPI\_Reduce}(\oplus); \end{array} \right] \Longrightarrow \left[ \begin{array}{l} \text{Make\_pair}; \\ \text{MPI\_Reduce}(\langle \oplus, \otimes \rangle); \\ \text{if my\_pid==ROOT then Take\_first}; \end{array} \right]$$

Here, functions `Make_pair` and `Take_first` implement *map pair* and  $\pi_1$  of (7), respectively. The result of reduction in MPI is a single datum stored in the processor whose `my_pid = ROOT`. Note that the scan-reduce transformation is applicable directly to our example program (1), fusing two collective operations into one, with simple local computations beforehand and afterward.

Our next rule transforms a composition of two scans [7]:

**Theorem 2 (Scan-Scan Composition ).** *For arbitrary binary associative operators  $\oplus$  and  $\otimes$ , if operator  $\otimes$  distributes over  $\oplus$ , then*

$$\text{scan}(\oplus) \circ \text{scan}(\otimes) = \text{map } \pi_1 \circ \text{scan}(\langle \oplus, \otimes \rangle) \circ \text{map pair} \quad (11)$$

where the functions *pair*,  $\pi_1$ ,  $\langle \oplus, \otimes \rangle$  are defined by (8)–(10).

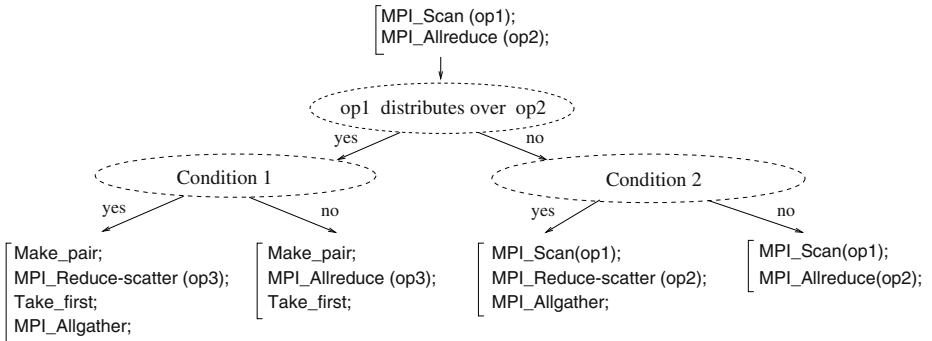
The theorem has the following MPI formulation:

$$\left[ \begin{array}{l} \text{MPI\_Scan}(\otimes); \\ \text{MPI\_Scan}(\oplus); \end{array} \right] \Rightarrow \left[ \begin{array}{l} \text{Make\_pair}; \\ \text{MPI\_Scan}(\langle \oplus, \otimes \rangle); \\ \text{Take\_first}; \end{array} \right]$$

functions `Make_pair` and `Take_first` being explained in Theorem 1.

The composition rules provided by Theorems 1 and 2 have the following important properties: a) their correctness is proved formally; b) they are parameterized by  $\oplus$  and  $\otimes$ , and are thus valid for a variety of particular base operators; c) they can be applied independently of the architecture of the target parallel machine; d) they are valid for all possible implementations of the collective operations involved, i.e. different middleware mechanisms used by MPI.

These and other compositions rules can be combined in the program design process, e.g. we capture the design alternatives as a decision tree in Figure 2 [8]:



**Fig. 2.** Design alternatives for transforming a composition of scan and reduction

The best domain-specific solution is obtained by checking the conditions in the inner tree nodes. Conditions depend either on the problem properties (e.g. distributivity) or on the characteristics of the target machine (number of processors, etc.), the latter being expressed by **Condition 1** and **Condition 2**.

## 4 Domain-Specific Implementation of Compositions

In this section, we study a novel skeleton, called double-scan (DS), which is a composition of two simpler scan skeletons. We first study a generic parallel implementation of the DS skeleton, demonstrate that it is not cost-optimal, and then develop an alternative, cost-optimal implementation.

### 4.1 The Double-Scan (DS) Skeleton

Let us introduce some more skeletons, defined on non-empty lists as follows:

- Scan-left and scan-right: Computing prefix sums of a list by traversing the list from left to right (or vice versa) and applying a binary operator  $\oplus$ :

$$\text{scanl}(\oplus)([x_1, \dots, x_n]) = [x_1, (x_1 \oplus x_2), \dots, (\dots(x_1 \oplus x_2) \oplus x_3) \oplus \dots \oplus x_n]$$

$$\text{scanr}(\oplus)([x_1, \dots, x_n]) = [(x_1 \oplus \dots \oplus (x_{n-2} \oplus (x_{n-1} \oplus x_n) \dots)), \dots, x_n]$$

- Zip: Component-wise application of a binary operator  $\oplus$  to a pair of lists of equal length (in Haskell, a similar function is called `zipWith`):

$$\text{zip}(\oplus)([x_1, \dots, x_n], [y_1, \dots, y_n]) = [(x_1 \oplus y_1), \dots, (x_n \oplus y_n)]$$

A new skeleton, *double-scan* (DS) [9], is convenient for expressing particular applications. Both versions of the DS skeleton have two parameters, which are the base operators of their constituent scans.

**Definition 1.** For binary operators  $\oplus$  and  $\otimes$ , two versions of the double-scan (DS) skeleton are defined as compositions of two scans, one left and one right:

$$\text{scanrl}(\oplus, \otimes) = \text{scanr}(\oplus) \circ \text{scanl}(\otimes) \quad (12)$$

$$\text{scanlr}(\oplus, \otimes) = \text{scanl}(\oplus) \circ \text{scanr}(\otimes) \quad (13)$$

The DS skeleton is a composition of two scans. In the previous section, we demonstrated that if both scan operators are associative, then a) both scans can be parallelized, and b) the composition can be fused according to Theorem 2. Here, we are interested in the situation where  $\oplus$  is non-associative.

### 4.2 Implementation of DS Using DH

Our recent result [9] provides the sufficient condition under which the double-scan skeleton can be expressed using a specific divide-and-conquer skeleton called DH (distributable homomorphism):

**Theorem 3.** Let  $\textcircled{1}$ ,  $\textcircled{2}$ ,  $\textcircled{3}$ ,  $\textcircled{4}$  be binary operators, where  $\textcircled{2}$  and  $\textcircled{4}$  are associative ( $\textcircled{1}$ ,  $\textcircled{3}$  need not be associative). If the following equality holds:

$$\text{scanrl}(\textcircled{1}, \textcircled{2}) = \text{scanlr}(\textcircled{3}, \textcircled{4}) \quad (14)$$

then  $\text{scanrl}(\textcircled{1}, \textcircled{2})$  can be represented as follows:

$$\text{scanrl}(\textcircled{1}, \textcircled{2}) = \text{scanlr}(\textcircled{3}, \textcircled{4}) = \text{map}(\pi_1) \circ dh(\oplus, \otimes) \circ \text{map}(\text{triple}) \quad (15)$$

where  $dh$  is the DH skeleton defined for arbitrary lists  $x$  and  $y$  [10]:

$$\begin{aligned} dh(\oplus, \otimes)[a] &= [a], \\ dh(\oplus, \otimes)(x \mathbin{++} y) &= \text{zip}(\oplus)(dh\ x, dh\ y) \mathbin{++} \text{zip}(\otimes)(dh\ x, dh\ y) \end{aligned} \quad (16)$$

with the following base operators  $\oplus$  and  $\otimes$ :

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \oplus \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 \textcircled{1} (a_3 \textcircled{2} b_2) \\ a_2 \textcircled{1} (a_3 \textcircled{2} b_2) \\ (a_3 \textcircled{4} b_2) \textcircled{3} b_3 \end{pmatrix} \quad \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \otimes \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} (a_3 \textcircled{4} b_2) \textcircled{3} b_1 \\ a_2 \textcircled{1} (a_3 \textcircled{2} b_2) \\ (a_3 \textcircled{4} b_2) \textcircled{3} b_3 \end{pmatrix}$$

The DH skeleton is a special form of the well-known divide-and-conquer paradigm: to compute  $dh$  on a concatenation of two lists,  $x \mathbin{++} y$ , we apply  $dh$  to  $x$  and  $y$ , then combine the results element-wise using  $\text{zip}$  with the parameter operators  $\oplus$  and  $\otimes$  and concatenate them. For this skeleton, there exists a family of generic parallel implementations, directly expressible in MPI [11].

Let us now turn to the parallel implementation of skeletons and its runtime. Basic skeletons have an obvious data-parallel semantics; the parallel time complexity is constant for  $\text{map}$  and  $\text{zip}$  and logarithmic for  $\text{scan}$  if  $\oplus$  is associative. If  $\oplus$  is non-associative, then  $\text{scan}$  has linear time complexity.

The *generic DH implementation* splits the input list into  $p$  blocks of approximately the same size, computes DH locally in the processors, and then performs  $\log p$  rounds of pairwise communications and computations in a hypercube-like manner [10]. Its time complexity is  $\Theta(t_{\text{seq}}(m) + m \cdot \log p)$ , where  $t_{\text{seq}}(m)$  is the time taken to sequentially compute DH on a block of length  $m \approx n/p$ . There always exists an obvious sequential implementation of the DH on a data block of size  $m$ , which has a time complexity of  $\Theta(m \cdot \log m)$ . Thus, in the general case, the time complexity of the generic DH implementation is  $\Theta(n/p \cdot \max\{\log(n/p), \log p\})$ .

Our first question is whether the implementation of DS based on DH is cost-optimal. The following theorem shows how the cost optimality of the generic DH implementation, used for a particular instance of the DH skeleton, depends on the optimal sequential complexity of this instance, i.e. particular application:

**Theorem 4.** *The generic parallel implementation of DH, when used for a DH instance with optimal sequential time complexity of  $t_{\text{seq}}$ , is*

- a) *cost-optimal on  $p \in O(n)$  processors, if  $t_{\text{seq}}(n) \in \Theta(n \cdot \log n)$ , and*
- b) *non-cost-optimal on  $p \in \omega(1)$  processors, if  $t_{\text{seq}}(n) \in o(n \cdot \log p)$ .*

Here is the proof sketch: a)  $c_p \in \Theta(n \cdot \max\{\log(n/p), \log p\}) \subseteq O(n \cdot \log n)$ , and b)  $c_p \in \Theta(n \cdot \max\{\log(n/p), \log p\}) \subseteq \Omega(n \cdot \log p)$ .

Since sequential time complexity of the DS skeleton as a composition of two scans is obviously linear, it follows from Theorem 4(b) that the generic DH-implementation cannot be cost-optimal for the DS skeleton.

This general result is quite disappointing. However, since there are instances of the DS skeleton that do have cost-optimal hand-coded implementations, we can strive to find a generic, cost-optimal solution for all instances. This motivates our further search for a better parallel implementation of DS.

### 4.3 Towards a Cost-Optimal Implementation of DS

In this section, we seek a cost-optimal implementation of the DS skeleton which is a composition of two scans. We introduce a special intermediate data structure, *pointed lists* (*plists*). A *k-plist*, where  $k > 0$ , consists of  $k$  conventional lists, called segments, and  $k - 1$  points between the segments:

$$\begin{array}{ccccccc} l_1 & a_1 & l_2 & a_2 & l_3 & a_3 & \dots & a_{k-1} & l_k \\ \hline & \bullet & & \bullet & & \bullet & & \bullet & \end{array}$$

If parameter  $k$  is irrelevant, we simply speak of a plist instead of a  $k$ -plist. Conventional lists are obviously a special case of plists. To distinguish between functions on lists and plists, we prefix the latter with the letter  $p$ , e. g. *pmap*. To transform between lists and plists, we use the following two functions:

- *list2plist<sub>k</sub>* transforms (partitions) an arbitrary list into a plist, consisting of  $k$  segments and  $k - 1$  points:

$$list2plist_k(l_1 \uplus [a_1] \uplus \dots \uplus [a_{k-1}] \uplus l_k) = [l_1, a_1, \dots, a_{k-1}, l_k]$$

Our further considerations are valid for arbitrary partitions but, in parallel programs, one usually works with segments of approximately the same size.

- *plist2list<sub>k</sub>*, the inverse of *list2plist<sub>k</sub>*, transforms a  $k$ -plist into a list:

$$plist2list([l_1, a_1, \dots, a_{k-1}, l_k]) = l_1 \uplus [a_1] \uplus \dots \uplus [a_{k-1}] \uplus l_k$$

Our goal is to provide a parallel implementation for a distributed version of *scanrl*, function *pscanrl*, which computes *scanrl* on a plist, so that the original function is yielded by the following composition of skeletons:

$$scanrl(\textcircled{1}, \textcircled{2}) = plist2list \circ pscanrl(\textcircled{1}, \textcircled{2}) \circ list2plist_k \quad (17)$$

We introduce the following auxiliary skeletons as higher-order functions on plists, omitting their formal definitions and illustrating them graphically instead:

- *pmap<sub>l</sub>*  $g$  applies function  $g$ , defined on lists, to all segments of a plist

$$\begin{array}{ccccccc} g & & g & & g \\ \hline & \bullet & \dots & \bullet & & \bullet & \dots & \bullet \\ & a_1 & & a_i & & a_{i+1} & & a_{k-1} \end{array}$$

- *pscanrl<sub>p</sub>*  $(\oplus, \otimes)$  applies function *scanrl*  $(\oplus, \otimes)$ , defined by (12), to the list containing only the points of the argument plist

$$\begin{array}{ccccccc} \hline & \bullet & & \bullet & & \bullet & \dots & \bullet \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & \\ scanrl(\oplus, \otimes) [ & \bullet & & \bullet & & \bullet & \dots & \bullet ] \end{array}$$

- $pinmap\_l(\odot, \oplus, \otimes)$  modifies each segment of a plist using operation  $\oplus$  for the left-most segment,  $\otimes$  for the right-most segment, and  $\odot$  for all inner segments, where operation  $\odot$  is a ternary operator, i.e. it gets a pair of points and a single point as parameters:

$$\frac{map(\oplus a_1)}{a_1} \bullet \dots \bullet \frac{map((a_i, a_{i+1}) \odot)}{a_{i+1}} \bullet \dots \bullet \frac{map(a_{k-1} \otimes)}{a_{k-1}}$$

- $pinmap\_p(\oplus, \otimes)$  modifies each single point of a plist depending on the last element of the point's left neighbouring segment and the first element of the point's right neighbouring segment:

$$\frac{}{l_1} \dots \frac{\overbrace{(last(l_i) \oplus a_i) \otimes first(l_{i+1})}}{l_i \bullet l_{i+1}} \dots \frac{}{l_k}$$

The next theorem shows that the distributed version of the double-scan skeleton can be implemented using our auxiliary skeletons. We use the following new definition: a binary operation  $\otimes$  is said to be *associative modulo*  $\odot$ , iff for arbitrary elements  $a, b, c$  we have:  $(a \otimes b) \otimes c = (a \odot b) \otimes (b \otimes c)$ . The usual associativity is the associativity modulo operation *first*, which yields the first element of a pair.

**Theorem 5.** ([12]) *Let  $\textcircled{1}$ ,  $\textcircled{2}$ ,  $\textcircled{3}$  and  $\textcircled{4}$  be binary operators, such that  $\textcircled{1}$  and  $\textcircled{3}$  are associative,  $scanrl(\textcircled{1}, \textcircled{2}) = scanlr(\textcircled{3}, \textcircled{4})$ , and  $\textcircled{2}$  is associative modulo  $\textcircled{4}$ . Moreover, let  $\textcircled{5}$  be a ternary operator, such that  $(a, a \textcircled{3} c) \textcircled{5} b = a \textcircled{3} (b \textcircled{1} c)$ . Then, the double-scan skeleton  $pscanrl$  on plists can be implemented as follows:*

$$pscanrl(\textcircled{1}, \textcircled{2}) = pinmap\_l(\textcircled{5}, \textcircled{1}, \textcircled{3}) \circ pscanrl\_p(\textcircled{1}, \textcircled{2}) \quad (18) \\ \circ pinmap\_p(\textcircled{2}, \textcircled{4}) \circ (pmap\_l \textcircled{scanrl}(\textcircled{1}, \textcircled{2}))$$

Since  $pscanrl(\textcircled{1}, \textcircled{2}) = pscanlr(\textcircled{3}, \textcircled{4})$ , the equality (18) also holds for the function  $pscanlr(\textcircled{3}, \textcircled{4})$ . There is a constructive procedure for generating the operator  $\textcircled{5}$ , if  $(a \textcircled{3})$  is bijective for arbitrary  $a$  and if  $\textcircled{3}$  distributes over  $\textcircled{1}$ .

On a parallel machine, we split plists so that each segment and its right “border point” are mapped to a processor. The time complexity analysis [9] shows that the DS implementation on plists is cost-optimal.

**Theorem 6.** *The parallel implementation (18) of the double-scan skeleton on plists is cost-optimal if  $p \in O(n/\log n)$  processors are used.*

The theorem ensures the cost optimality of the suggested parallel implementation in the practice-relevant case when the number of processors is not too big. The estimate  $O(n/\log n)$  provides an upper bound for how fast the number of processors is allowed to grow with the problem size. If the number of processors grows slower than the bound, the implementation is cost optimal.

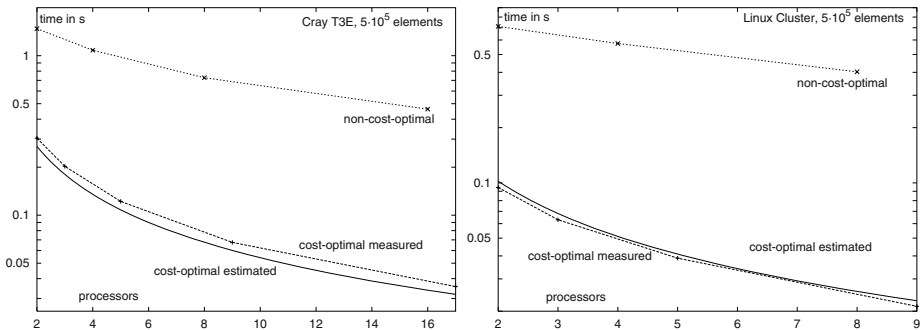
#### 4.4 Case Study: Tridiagonal System Solver

By way of an example application, we consider the solution of a tridiagonal system of equations [9]. The given system is represented as a list of rows consisting of four values: the value on the main, upper and lower diagonal of the matrix and the value of the right-hand-side vector, respectively.

A popular algorithm for solving a tridiagonal system is Gaussian elimination [2, 3], which eliminates the lower and upper diagonal of the matrix in two steps: 1) eliminates the lower diagonal by traversing the matrix from top-down using the *scanl* skeleton with an operator  $\textcircled{2}_{tds}$  [9]; 2) eliminates the upper diagonal by a bottom-up traversal, i.e. using the *scanr* with an operator  $\textcircled{1}_{tds}$ . We can alternatively eliminate first the upper and then the lower diagonal using two other row operators,  $\textcircled{3}_{tds}$  and  $\textcircled{4}_{tds}$ .

Being a composition of two scans, the tridiagonal system solver is a natural candidate for treatment as an instance of the DS skeleton. Since the conditions of Theorem 5 are satisfied [12], our new cost-optimal DS implementation (17)–(18) can be used for the tridiagonal solver. In this case, the implementation operates on lists of rows, which are quadruples of double values.

We conducted experiments with the tridiagonal system solver, based on the DS skeleton, on two different machines: 1) a Cray T3E machine with 24 processors of type Alpha 21164, 300 MHz, 128 MB, using native MPI implementation, and 2) a Linux cluster with 16 nodes, each consisting of two processors of type Pentium IV, 1.7 GHz, 1 GB, using an SCI port of MPICH.



**Fig. 3.** Runtimes of the tridiagonal solver: cost-optimal vs. non-cost-optimal implementation vs. predicted runtime. Left: on the Cray T3E; right: on the Linux cluster.

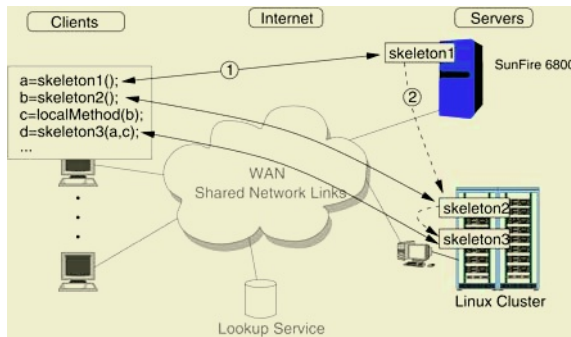
In Fig. 3, the cost-optimal version demonstrates a substantial improvement of the runtime compared with the generic, non-cost-optimal version: up to 13 on 17 processors of the Cray T3E and up to 18 on 9 processors of the cluster. Note that we use a logarithmic scale for the time axis. In Fig. 3, we also compare the analytically predicted runtime with the measured times for a fixed problem size. The quality of prediction is quite good: the difference to the measured times is less than 12% on both machines.

## 5 Compositions of Remote Skeletons on the Grid

In this section, we consider the use of program components in general and skeletons in particular in a distributed environment. Our systems of interest are so-called computational grids that connect high-performance computational servers via the Internet and make them available to application programmers.

### 5.1 Grid Programming Using Skeletons

Figure 4 shows an exemplary Grid-like system used in our experiments. There are two high-performance servers: a shared-memory multiprocessor SunFire 6800 and a Linux cluster. Application programmers work from clients (PCs and workstations). A central entity called “lookup service” is used for resource discovery. The reader is referred to [13] for details of the system architecture and the issues of resource discovery and management.



**Fig. 4.** System architecture and interaction of its parts

We develop application programs for such Grid systems using skeletons. As shown in the figure, time-intensive skeleton calls are executed remotely on servers that provide implementations for the corresponding skeleton (arrow ①). If two subsequent skeleton calls are executed on different servers, then the result of the first call must be communicated as one of inputs for the second call (arrow ②). Thus, we are dealing again with the sequential composition of parallel skeletons.

Using skeletons for programming on the Grid has the following advantages:

- The skeletons’ implementations on the server side are usually highly efficient because they can be carefully tuned to the particular server architecture.
- The once-developed, provably correct implementation of a skeleton on a particular server can be reused by different applications.
- Skeletons hide details about the executing hardware and the server’s communication topology from the application programmer.
- Skeletons provide a reliable model for performance prediction, offering a sound information base for selecting servers.



Our skeleton-based Grid programming environment is built on top of Java and RMI. We chose the Java platform mostly for reasons of portability (see “10 reasons to use Java in Grid computing” [14]).

In the system, skeletons are offered as Java (remote) interfaces, which can be implemented in different ways on different servers. Functional parameters are provided as codes, in our Grid system using Java bytecode. To be as flexible as possible, all skeletons operate on `Object`s or arrays of `Object`. For example, the interface for the `scan` skeleton contains a single method

```
public Object[] invoke(Object[], BinOp op);
```

To use the `scan` skeleton, the client first finds a server for execution using the lookup service [13]. After obtaining an RMI reference to the `scan` implementation on the server, the skeleton is executed via RMI by calling the `invoke` method with appropriate parameters.

## 5.2 Domain-Specific Compositions: Future-Based RMI

Using the Java RMI mechanism in distributed programs with skeletons has the important advantage that the outsourcing of skeleton calls to remote servers is invisible for the programmer: remote calls are coded in exactly the same way as local calls. However, since the RMI mechanism was developed for client-server systems, it is not optimal for the Grid. We illustrate this using a composition of two skeleton calls, with the result of the first call being used as an argument of the second call (`skeleton1` and `skeleton2` are remote references):

```
result1 = skeleton1.invoke(...);
result2 = skeleton2.invoke(result1, ...);
```

When executing such a composition of methods using standard RMI, the result of a remote method invocation is always sent back directly to the client. This is illustrated for the above example in Fig. 5 (left). When `skeleton1` is invoked (①), the result is sent back to the client (②), and then to `skeleton2` (③). Finally, the result is sent back to the client (④). For typical applications consisting of many composed skeletons, this feature of RMI results in very high time overhead.

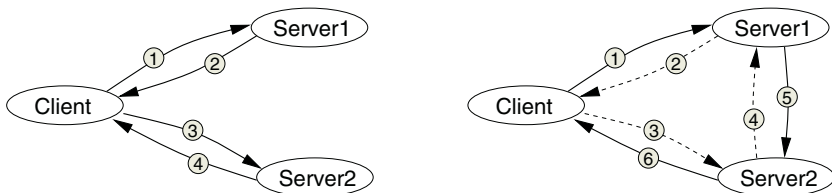


Fig. 5. Skeleton composition using plain RMI (left) and future-based RMI (right)

To eliminate this overhead, we have developed so-called *future-based RMI*: an invocation of a skeleton on a server initiates the skeleton’s execution and then returns immediately, without waiting for the skeleton’s completion (see

Figure 5, right). As a result of the skeleton call, a future remote reference is returned to the client (②), rather than the result itself. This reference is then used as the parameter for invoking the next skeleton (③). When the future reference is dereferenced (④), the dereferencing thread on the server is blocked until the result is available, i.e. the first skeleton actually completes. The result is then sent directly to the server dereferencing the future reference (⑤). After completion of `skeleton2`, the result is sent to the client (⑥).

Compared with plain RMI, our future-based mechanism substantially reduces the amount of data sent over the network, because only a reference to the data is sent to the client; the result itself is communicated directly between the servers. Moreover, communications and computations overlap, effectively hiding latencies of remote calls. We have implemented future-based RMI on top of SUN RMI.

Future references are available to the user through a special Java interface `RemoteReference`. There are only a few minor differences when using future-based RMI compared with the use of plain RMI: 1) instead of `Objects`, all skeletons return values of type `RemoteReference`, and 2) the skeletons' interfaces are extended by `invoke` methods, accepting `RemoteReferences` as parameters.

### 5.3 Application Case Study and Measurements

We have tested our approach on a linear system solver, which we implement using the matrix library Jama [15]. For a given system  $Ax = b$ , we look for vector  $\hat{x}$  that minimizes  $\chi^2 = (Ax - b)^2$ . Our example uses the singular value decomposition method (SVD): the SVD of a matrix is a decomposition  $A = U \cdot \Sigma \cdot V^T$ ,  $U$  and  $V$  being orthonormal matrices and  $\Sigma$  a diagonal matrix [16].

```
RemoteReference rA,rx,rb,U,S,V,err,svd,rr;
Matrix A,b,x; SVD svd; double r;
rA=new RemoteReference(A); rb=new RemoteReference(b);
//decompose
svd = srv.svd(rA); U = srv.s_getU(svd);
S = srv.s_getS(svd); V = srv.s_getV(svd);
S=srv.execTask(new Adjust(),S); //adjust
//compute result
rx=srv.times(V,srv.times(S,srv.times(srv.transpose(U),rb)));
err = srv.minus(srv.times(rA,x),rb);
rr = srv.normInf(err); x = rx.getValue(); r = rr.getValue();
```

**Fig. 6.** Case study: composition of remote methods using future-based RMI

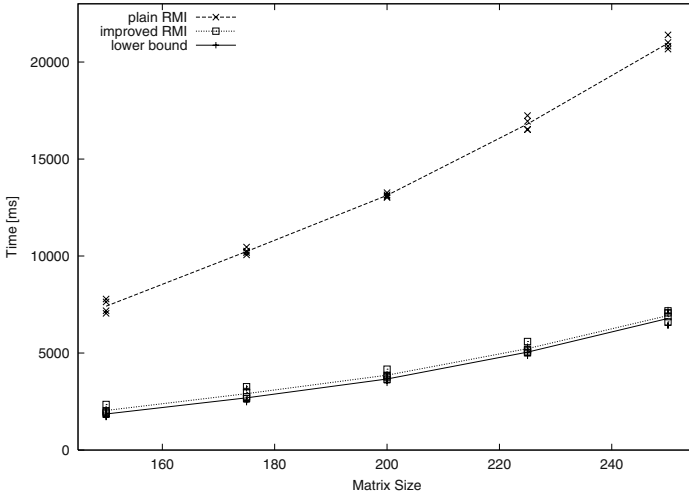
The solver is a composition of four steps. First the SVD for  $A$  is computed, using a Jama library method. Then the inverse of  $A$  is computed, and  $A^{-1}$  is multiplied by  $b$  to obtain  $x$ . Finally, the residual  $r = |A \cdot x - b|$  is computed.

The version of the program, with the composition expressed using the future-based RMI, is shown in Figure 6. The `srv` variable holds the RMI reference to a remote object on the server, providing access to the Jama methods. The

application-specific `adjust` method is called on the server by encapsulating the method in its own object (of type `Adjust`), which is executed on the server by providing it as a parameter to the server-sided method `execTask`. This improves performance because it takes more time to send matrix  $S$  over the network and compute on the client than to send the code to the server and execute it there.

Comparing future-based and plain RMI, the difference from the programmer's viewpoint is that parameters  $A$  and  $b$  are packed into `RemoteReferences` for use with the server-sided Jama methods. Besides, the results  $x$  and  $r$  are returned as `RemoteReferences` (`rr` and `rx`), and the `getValue()` method needs to be called to obtain the results.

Our testbed environment consists of two LANs, one at the TU Berlin and the other at the University of Erlangen, at a distance of approx. 500 km. We used a SunFire6800 as the server in Berlin and an UltraSparc Ii 360 MHz as the client in Erlangen, both with SUN JDK1.4.1 (HotSpot Client VM, mixed mode). Figure 7 shows the runtimes for three versions of the solver: plain RMI, future-based RMI, and the version running completely on the server side ("ideal").



**Fig. 7.** Measured runtimes for the SVD-solver

The plain RMI version is much slower (three to four times) than the "ideal" version, which runs completely on the server side. The improved RMI version is less than 10 % slower than the ideal version, so it eliminates most of the overhead of plain RMI.

## 6 Conclusions and Related Work

The desire to be able to name and reuse "programming patterns", i. e. to capture them in the form of parameterizable abstractions, has been a driving force in the evolution of high-level programming languages in general. In the sequential setting, design patterns [17] and components [18] are recent examples of this.

In parallel programming, where algorithmic aspects have traditionally been of special importance, the approach using algorithmic skeletons [1] has emerged.

The success of skeleton-based programming will hinge on the creation of a science to specify and optimize domain-specific programs that use skeletons. The key property of compositionality should be formulated and studied using suitable algebraic models. In the domain of parallel and distributed skeletons, the question arises whether the corresponding algebra is closed. The experience with relational algebra for databases indicates that the algebra of skeletons for general-purpose programming will probably never be closed nor complete.

Our work demonstrates that compositionality of components in parallel and distributed programming is not straightforward: it cannot be guaranteed that the sequential composition of the best parallel implementations of two components yields again optimal parallelism. Instead, a domain-specific implementation effort for compositions is often called for. We have demonstrated three abstraction levels at which domain-specific optimizations can be developed.

Therefore, it is not only important to provide useful program components with efficient individual implementations. In addition, rules and mechanisms for composition like our rules for collective operations and skeletons, should be developed which are suitable for envisaged target machines. Our composition rules are directly applicable both for skeletons and MPI collective operations. A similar approach is also pursued in the setting of functional programming [19].

In the domain of distributed computational grids, we address the problem of efficient skeleton composition at the level of middleware. We propose a novel, future-based RMI mechanism which reduces communication overhead when compositions of skeleton calls are executed. It differs from comparable approaches because it combines hiding network latencies using asynchronous methods [20, 21] and reducing network dataflow by allowing server/server communication [22]. Our approach is different from the pointer-chasing mechanism used in object migration systems [6]: future-based references can be used whenever an object (the result of a method) is needed on another server, which is not known during object creation.

## Acknowledgements

This work is based on results obtained in joint research with Christian Lengauer, Holger Bischof, Martin Alt, Christoph Wedler and Emanuel Kitzelmann, to all of whom the author is grateful for their cooperation. Anonymous referees and Julia Kaiser-Mariani helped a lot to improve the presentation.

## References

1. Cole, M.I.: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. PhD thesis, University of Edinburgh (1988)
2. Quinn, M.J.: Parallel Computing. McGraw-Hill, Inc. (1994)

3. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann Publ. (1992)
4. Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann Publ. (1997)
5. Gorlatch, S.: Message passing without send-receive. *Future Generation Computer Systems* **18** (2002) 797–805
6. Grune, D., Bal, H.E., Jacobs, C.J.H.: *Modern Compiler Design*. John Wiley (2000)
7. Gorlatch, S.: Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MIP-9711, Universität Passau (1997) Available at <http://www.fmi.uni-passau.de/cl/papers/Gor97b.html>.
8. Gorlatch, S.: Towards formally-based design of message passing programs. *IEEE Trans. on Software Engineering* **26** (2000) 276–288
9. Bischof, H., Gorlatch, S.: Double-scan: Introducing and implementing a new data-parallel skeleton. In Monien, B., Feldmann, R., eds.: *Euro-Par 2002*. Volume 2400 of LNCS., Springer (2002) 640–647
10. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y., eds.: *Euro-Par'96: Parallel Processing*, Vol. II. *Lecture Notes in Computer Science* 1124. Springer-Verlag (1996) 401–408
11. Gorlatch, S., Lengauer, C.: Abstraction and performance in the design of parallel programs: overview of the SAT approach. *Acta Informatica* **36** (2000) 761–803
12. Bischof, H., Gorlatch, S., Kitzelmann, E.: The double-scan skeleton and its parallelization. Technical Report 2002/06, Technische Universität Berlin (2002)
13. Alt, M., Bischof, H., Gorlatch, S.: Algorithm design and performance prediction in a Java-based Grid system with skeletons. In Monien, B., Feldmann, R., eds.: *Euro-Par 2002*. Volume 2400 of *Lecture Notes in Computer Science*., Springer (2002) 899–906
14. Getov, V., von Laszewski, G., Philippsen, M., Foster, I.: Multiparadigm communications in Java for Grid computing. *Communications of the ACM* **44** (2001) 118–125
15. Hicklin, J., Moler, C., Webb, P., Boisvert, R.F., Miller, B., Pozo, R., Remington, K.: JAMA: A Java matrix package. (<http://math.nist.gov/javanumerics/jama/>)
16. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: *Numerical Recipes in C: The Art of Scientific Computing*. Second edn. Cambridge University Press (1992)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. Addison Wesley (1995)
18. Szyperski, C.: *Component software: beyond object-oriented programming*. Addison Wesley (1998)
19. Hu, Z., Iwasaki, H., Takeichi, M.: An accumulative parallel skeleton for all. In: 11th European Symposium on Programming (ESOP 2002). *Lecture Notes in Computer Science* 2305. Springer (2002) 83–97
20. Raje, R., Williams, J., Boyles, M.: An asynchronous remote method invocation (ARMI) mechanism in Java. *Concurrency: Practice and Experience* **9** (1997) 1207–1211
21. Falkner, K.K., Coddington, P., Oudshoorn, M.: Implementing asynchronous remote method invocation in Java. In: *Proc. of Parallel and Real Time Systems (PART'99)*, Melbourne (1999) 22–34
22. Yeung, K.C., Kelly, P.H.J.: Optimising Java RMI programs by communication restructuring. In Schmidt, D., Endler, M., eds.: *Middleware 2003: ACM/I-FIP/USENIX International Middleware Conference*, Springer Verlag (2003) 324–343

# Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation

Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H.J. Kelly

Department of Computing, Imperial College London  
180 Queen's Gate, London SW7 2BZ, UK  
`{o.beckmann,p.kelly}@imperial.ac.uk`  
`www.doc.ic.ac.uk/{~ob3,~phjk}`

**Abstract.** The TaskGraph Library is a C++ library for dynamic code generation, which combines specialisation with dependence analysis and loop restructuring. A TaskGraph represents a fragment of code which is constructed and manipulated at runtime, then compiled, dynamically linked and executed. TaskGraphs are initialised using macros and overloading, which forms a simplified, C-like sub-language with first-class arrays and no pointers. Once a TaskGraph has been constructed, we can analyse its dependence structure and perform optimisations. In this Chapter, we present the design of the TaskGraph library, and two sample applications to demonstrate its use for runtime code specialisation and restructuring optimisation.

## 1 Introduction

The work we describe in this Chapter is part of a wider research programme at Imperial College aimed at addressing the apparent conflict between the quality of scientific software and its performance. The TaskGraph library, which is the focus of this Chapter, is a key tool which we are developing in order to drive this research programme. The library is written in C++ and is designed to support a model of software components which can be composed dynamically, and optimised, at runtime, to exploit execution context:

- *Optimisation with Respect to Runtime Parameters*

The TaskGraph library can be used for specialising software components according to either their parameters or other runtime context information. Later in this Chapter (Sec. 3), we show an example of specialising a generic image filtering function to the particular convolution matrix being used.

- *Optimisation with Respect to Platform*

The TaskGraph library uses SUIF-1 [1], the Stanford University Intermediate Format, as its internal representation for code. This makes a rich collection of dependence analysis and restructuring passes available for our use in code optimisation. In Sec. 5 of this Chapter we show an example of generating, at runtime, a matrix multiply component which is optimally tiled with respect to the host platform.

The TaskGraph library is a tool for implementing domain-specific optimisations in active libraries for high-performance computing; we discuss its relationship with the other work in this book in Sec. 8, which concludes the Chapter.

*Background.* Several earlier tools for dynamic code optimisation have been reported in the literature [2, 3]. The key characteristics which distinguish our approach are as follows:

- *Single-Language Design*

The TaskGraph library is implemented in C++ and any TaskGraph program can be compiled as C++ using widely-available compilers. This is in contrast with approaches such as Tick-C [2] which rely on a special compiler for processing dynamic constructs. The TaskGraph library’s support for manipulating code as data within one language was pioneered in Lisp [4].

- *Explicit Specification of Dynamic Code*

Like Tick-C [2], the TaskGraph library is an imperative system in which the application programmer has to construct the code as an explicit data structure. This is in contrast with ambitious partial evaluation approaches such as DyC [3, 5] which use declarative annotations of regular code to specify where specialisation should occur and which variables can be assumed constant. Offline partial evaluation systems like these rely on *binding-time analysis* (BTA) to find other, derived static variables [6].

- *Simplified C-like Sub-language*

Dynamic code is specified with the TaskGraph library via a small sub-language which is very similar to standard C (see Sec. 2). This language has been implemented through extensive use of macros and C++ operator overloading and consists of a small number of special control flow constructs, as well as special types for dynamically bound variables. Binding times of variables are explicitly determined by their C++ types, while binding times for intermediate expression values are derived using C++’s overloading mechanism (Sec. 4). The language has first-class arrays, unlike C and C++, to facilitate dependence analysis.

In Sec. 6 we discuss the relationship with other approaches in more detail.

*Structure of this Chapter.* In Sec. 2, we describe how TaskGraphs are constructed. Section 3 offers a simple demonstration of runtime specialisation. Section 4 explains how the library itself is implemented. In Sec. 5, we use matrix multiplication to illustrate the use of the library’s loop restructuring capabilities. In Secs. 6 and 7 we discuss related and ongoing work, and Sec. 8 concludes.

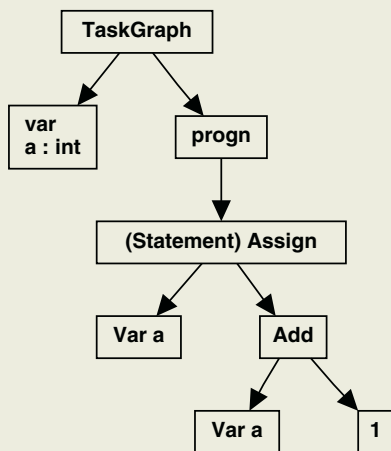
## 2 The TaskGraph Library API

A TaskGraph is a data structure which holds the abstract syntax tree (AST) for a piece of dynamic code. A key feature of our approach is that the application programmer has access to and can manipulate this data structure at

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <TaskGraph>
4 using namespace tg;
5 int main( int argc, char *argv[] ) {
6     TaskGraph T;
7     int b = 1;
8     int c = atoi( argv[1] );
9     taskgraph( T ) {
10         tParameter( tVar( int, a ) );
11         a = a + c;
12     }
13     T.compile();
14     T.execute( "a", &b, NULL );
15     printf( "b=%d\n", b );
16 }

```



**Fig. 1.** *Left:* Simple Example of using the TaskGraph library. *Right:* Abstract syntax tree (AST) for the simple TaskGraph constructed by the piece of code shown on the left. The `int` variable `c`, is static at TaskGraph construction time, and appears in the AST as a value (see Sec. 4). The (not type-safe) `execute()` call takes a NULL-terminated list of parameter name/value pairs; it binds TaskGraph parameter “a” to the address of the integer `b`, then invokes the compiled code.

runtime; in particular, we provide an extensible API (sub-language) for *constructing* TaskGraphs at runtime. This API was carefully designed using macros and C++ operator overloading to look as much as possible like ordinary C.

*A Simple Example.* The simple C++ program shown in the left-hand part of Fig. 1 is a complete example of using the TaskGraph library. When compiled with `g++`, linked against the TaskGraph library and executed, this program dynamically creates a piece of code for the statement `a = a + c`, binds the application program variable `b` as a parameter and executes the code, printing `b = 2` as the result. This very simple example illustrates both that creation of dynamic code is completely explicit in our approach and that the language for creating the AST which a TaskGraph holds looks similar to ordinary C.

### 3 Generalised Image Filtering

We now show an example which uses a fuller range of TaskGraph constructs and which also demonstrates a real performance benefit from runtime code optimisation. A generic image convolution function, which allows the application programmer to supply an arbitrary convolution matrix could be written in C as shown in Fig. 2. This function has the advantage of genericity (the interface is in principle similar to the General Linear Filter functions from the Intel Performance Libraries [7, Sec. 9]) but suffers from poor performance because



```

void convolution( const int IMGsz, const float *image, float *new_image,
                 const int CSZ /* convolution matrix size */, const float *matrix ) {
    int i, j, ci, cj; const int c_half = ( CSZ / 2 );

    // Loop iterating over image
    for( i = c_half; i < IMGsz - c_half; ++i ) {
        for( j = c_half; j < IMGsz - c_half; ++j ) {
            new_image[i * IMGsz + j] = 0.0;

            // Loop to apply convolution matrix
            for( ci = - c_half; ci <= c_half; ++ci ) {
                for( cj = - c_half; cj <= c_half; ++cj ) {
                    new_image[i * IMGsz + j] += ( image[(i+ci) * IMGsz + j+cj] *
                                                    matrix[(c_half+ci) * CSZ + c_half+cj] );
                }
            }
        }
    }
}

```

**Fig. 2.** Generic image filtering: C++ code. Because the size as well as the entries of the convolution matrix are runtime parameters, the inner loops (for-ci and for-cj), with typically very low trip-count, cannot be unrolled efficiently.

- The bounds of the inner loops over the convolution matrix are statically unknown, hence these loops, with typically very low trip-count, cannot be unrolled efficiently.
- Failure to unroll the inner loops leads to unnecessarily complicated control flow and also blocks optimisations such as vectorisation on the outer loops.

Figure 3 shows a function which constructs a TaskGraph that is specialised to the particular convolution matrix being used. The `tFor` constructs are part of the TaskGraph API and create a loop node in the AST. Note, however, that the inner `for` loops are executed as ordinary C++ at TaskGraph *construction* time, creating an assignment node in the AST for each iteration of the loop body. The effect is that the AST contains control flow nodes for the for-i and for-j loops and a loop body consisting of  $CSZ * CSZ$  assignment statements.

We study the performance of this example in Fig. 4. The convolution matrix used was a  $3 \times 3$  averaging filter, images were square arrays of single-precision floats ranging in size up to  $4096 \times 4096$ . Measurements are taken on a Pentium 4-M with 512KB L2 cache running Linux 2.4, gcc 3.3 and the Intel C++ compiler version 7.1. We compare the performance of the following:

- The static C++ code, compiled with gcc 3.3 (-O3).
- The static C++ code, compiled with the Intel C++ compiler version 7.1 (-restrict -O3 -ipo -xiMKW -tpp7 -fno-alias). The icc compiler reports that the innermost loop (for-cj) has been vectorised<sup>1</sup>. Note, however, that this loop will have a dynamically determined trip-count of 3, i.e. the Pentium 4’s 16-byte vector registers will not be filled.
- The code dynamically generated by the TaskGraph library, compiled with gcc 3.3 (-O3). The two innermost loops are unrolled.

<sup>1</sup> The SSE2 extensions implemented on Pentium 4 processors include 16-byte vector registers and corresponding instructions which operate simultaneously on multiple operands packed into them [8].

```

1  void taskgraph_convolution( TaskGraph &T, const int IMGsz,
2                                const int CSZ, const float *matrix ) {
3      int ci , cj;
4      assert ( CSZ % 2 == 1 );
5      const int c_half = ( CSZ / 2 );
6
7      taskgraph( T ) {
8          unsigned int dims[] = {IMGsz * IMGsz};
9          tParameter( tArrayFromList( float, tging, 1, dims ) );
10         tParameter( tArrayFromList( float, new_tging, 1, dims ) );
11         tVar ( int, i );
12         tVar ( int, j );
13
14         // Loop iterating over image
15         tFor( i, c_half, IMGsz - (c_half + 1) ) {
16             tFor( j, c_half, IMGsz - (c_half + 1) ) {
17                 new_tging[i * IMGsz + j] = 0.0;
18
19                 // Loop to apply convolution matrix
20                 for( ci = -c_half; ci <= c_half; ++ci ) {
21                     for( cj = -c_half; cj <= c_half; ++cj ) {
22                         new_tging[i * IMGsz + j] +=
23                             tging[(i+ci) * IMGsz + j+cj] * matrix[(c_half+ci) * CSZ + c_half+cj];
24                     } } }
25             }
26     }

```

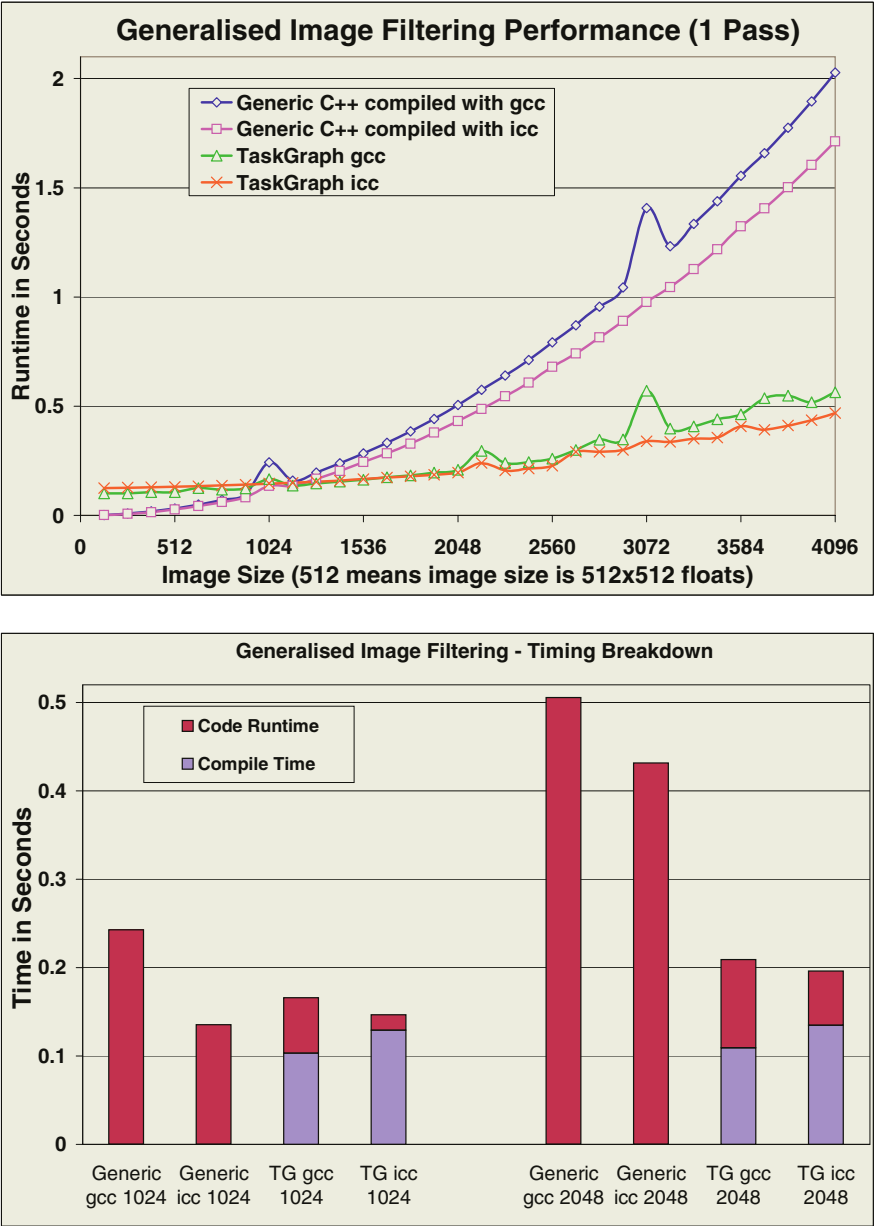
**Fig. 3.** Generic image filtering: function constructing the TaskGraph for a specific convolution matrix. The size as well as the entries of the convolution matrix are static at TaskGraph construction time. This facilitates complete unrolling of the inner two loops. The outer loops (for-i and for-j) are entered as control flow nodes in the AST.

- The code dynamically generated by the TaskGraph library, compiled with `icc 7.1 (-restrict -O3 -ipo -xiMKW -tpp7 -fno-alias)`. The two innermost loops are unrolled and the then-remaining innermost loop (the for-j loop over the image) is vectorised by `icc`.

In a realistic image filtering application, the datatype would probably be fixed-precision integers, or, in more general scientific convolution applications, double-precision floats. We chose single-precision floats here in order illustrate the fact that specialisation with respect to the bounds of the inner loop can facilitate efficient vectorisation.

We have deliberately measured the performance of these image filtering functions for only one pass over an image. In order to see a real speedup the overhead of runtime compilation therefore needs to be recovered in just a single application of the generated code. Figure 4 shows that we do indeed get an overall speedup for image sizes that are greater than  $1024 \times 1024$ . In the right-hand part of Fig. 4, we show a breakdown of the overall execution time for two specific data sizes. This demonstrates that although we achieve a huge reduction in execution time of the actual image filtering code, the constant overhead of runtime compilation cancels out this benefit for a data size of  $1024 \times 1024$ . However, for larger data sizes, we achieve an overall speedup.

Note, also, that image filters such as the one in this example might be applied either more than once to the same image or to different images – in either case,



**Fig. 4.** Performance of image filtering example. Top: Total execution time, including runtime compilation, for one pass over image. Bottom: Breakdown of total execution time into compilation time and execution time of the actual convolution code for two specific image sizes: 1024 × 1024 (the break-even point) and 2048 × 2048.

we would have to pay the runtime compilation overhead only once and will get higher overall speedups.

## 4 How It Works

Thus far, we have given examples of how the TaskGraph library is used, and demonstrated that it can achieve significant performance gains. In this section we now give a brief overview of TaskGraph syntax, together with an explanation of how the library works.

*TaskGraph Creation.* The TaskGraph library can represent code as data – specifically, it provides TaskGraphs as data structures holding the AST for a piece of code. We can create, compile and execute different TaskGraphs independently. Statements such as the assignment `a = a + c` in line 11 of Fig. 1 make use of C++ operator overloading to add nodes (in this case an assignment statement) to a TaskGraph. Figure 1 illustrates this by showing a graphical representation of the complete AST which was created by the adjacent code. Note that the variable `c` has *static* binding-time for this TaskGraph. Consequently, the AST contains its value rather than a variable reference.

The `taskgraph( T ){...}` construct (see line 7 in Fig. 3) determines which AST the statements in a block are attached to. This is necessary in order to facilitate independent construction of different TaskGraphs.

*Variables in TaskGraphs.* The TaskGraph library inherits lexical scoping from C++. The `tVar(type, name)` construct (see lines 11 and 12 in Fig. 3) can be used to declare a dynamic local variable.

Similarly, the `tArray(type, name, no_dims, extents[])` construct can be used to declare a dynamic multi-dimensional array with number of dimensions `no_dims` and size in each dimension contained in the integer array `extents`. Arrays are first-class objects in the TaskGraph construction sub-language and can only be accessed inside a TaskGraph using the `[]` subscript operators. There are no pointers in the TaskGraph construction sub-language.

*TaskGraph Parameters.* Both Fig. 1 (line 10) and Fig. 3 (lines 9 and 10) illustrate that any TaskGraph variable can be declared to be a TaskGraph parameter using the `tParameter()` construct. We require the application programmer to ensure that TaskGraph parameters bound at execution time do not alias each other.

*Control Flow Nodes.* Inside a TaskGraph construction block, `for` loops and `if` conditionals are executed at construction time. Therefore, the `for` loops on lines 20 and 21 in Fig. 3 result in an unrolled inner loop. However, the TaskGraph sub-language defines some constructs for adding control-flow nodes to an AST: `tFor(var, lower, upper)` adds a loop node (see lines 15 and 16 in Fig. 3). The loop bounds are *inclusive*. `tIf()` can be used to add a conditional node to the AST. The TaskGraph embedded sublanguage also includes `tWhile`, `tBreak` and `tContinue`. Function call nodes can be built, representing execution-time calls to functions defined in the host C++ program.

```

303 taskgraph( T ) {
304   unsigned int dims[] = {IMGSZ * IMGSZ};
305   tParameter( tArrayFromList( float, tging, 1, dims ) );
306   tParameter( tArrayFromList( float, new_tging, 1, dims ) );
307   tVar ( int, i );
308   tVar ( int, j );
309
310   // Loop iterating over image
311   tFor( i, c_half, IMGSZ - (c_half + 1) ) {
312     tFor( j, c_half, IMGSZ - (c_half + 1) ) {
313       new_tging[i * IMGSZ + j] = 0.0;
314
315       // Loop to apply convolution matrix
316       for( ci = -c_half; ci <= c_half; ++ci ) {
317         for( cj = -c_half; cj <= c_half; ++cj ) {
318           new_tging[i * IMGSZ + j] +=
319             tging[(i+ci) * IMGSZ + j+cj] * matrix[(c_half+ci) * CSZ + c_half+cj];
320         } } }
321   }
322 }

```

**Fig. 5.** Binding-Time Derivation. TaskGraph construction code for the image filtering example from Fig. 2, with all dynamic variables marked by a boxed outline.

*Expressions and Binding-Time.* We refer to variables that are bound at TaskGraph construction time as *static* variables and those that are bound at execution time as *dynamic*. Declarative code specialisation systems such as DyC [3] use annotations that declare some variables to be static for the purpose of partial evaluation. In contrast, static binding time, i.e. evaluated at TaskGraph construction time is the default for the TaskGraph language. Only TaskGraph variables and parameters are dynamic; this is indicated explicitly by declaring them appropriately (**tVar**, **tParameter**). The overloaded operators defined on those dynamic types define binding-time derivation rules. Thus, an expression such as  $a + c$  in Fig. 1 where  $a$  is dynamic and  $c$  is static is derived dynamic, but the static part is evaluated at construction time and entered into the AST as a value. We illustrate this by reproducing the TaskGraph image filtering code from Fig. 3 again in Fig. 5; however, this time all dynamic expressions are marked by a boxed outline. Note that the convolution matrix, including its entire subscript expression in the statement on line 22, is *static*.

## 5 Another Example: Matrix Multiply

In Sec. 3, we showed an example of how the specialisation functionality of the TaskGraph library can be used to facilitate code optimisations such as vectorisation. In this Section, we show, using matrix multiplication as an example, how we can take advantage of the use of SUIF-1 as the underlying code representation in the TaskGraph library to perform restructuring optimisations at runtime.

<pre> /*  * mm_ijk  * Most straight-forward matrix multiply  * Calculates C += A * B  */ void mm_ijk( const unsigned int sz,              const float *const A,              const float *const B,              float *const C ) {     unsigned int i, j, k;     for( i = 0; i &lt; sz; ++i ) {         for( j = 0; j &lt; sz; ++j ) {             for( k = 0; k &lt; sz; ++k ) {                 C[i*sz+j] += A[i*sz+k] * B[k*sz+j];             }         }     } } </pre>	<pre> void TG_mm_ijk( unsigned int sz[2],                 TaskLoopIdentifier *loop,                 TaskGraph &amp;t ) {     taskgraph( t ) {         tParameter(tArrayFromList(float,A,2,sz));         tParameter(tArrayFromList(float,B,2,sz));         tParameter(tArrayFromList(float,C,2,sz));         tVar(int,i); tVar(int,j); tVar(int,k);          tGetId( loop [0] );         tFor( i, 0, sz [0] - 1 ) {             tGetId( loop [1] );             tFor( j, 0, sz [1] - 1 ) {                 tGetId( loop [2] );                 tFor( k, 0, sz [0] - 1 ) {                     C[i][j] += A[i][k] * B[k][j];                 }             }         }     } } } } </pre>
<pre> for( int tsz = 4; tsz &lt;= min(362, matsz); ++tsz ) {     for( int i = 0; i &lt; samples; ++i ) {         unsigned int sizes[] = { matsz, matsz };         int trip3[] = { tsz, tsz, tsz };         TaskLoopIdentifier loop [3];         TaskGraph MM;          TG_mm_ijk( sizes, loop, MM );         interchangeLoops( loop[1], loop [2] ); // Interchange loops         tileLoop( 3, &amp;loop [0], trip3 ); // Tile inner two loops         MM.compile( TaskGraph::ICC, false );          tt3 = time_function ();         MM.setParameters( "A", A, "B", B, "C", C, NULL );         MM.execute();         tt3 = time_function() - tt3;          time[i] = time_to_seconds( tt3 );     }     time[samples] = mean( samples, time );     if( time[samples] &lt; best_time_gcc ) {         best_time_gcc = time[samples];         best_tsz_gcc = tsz;     } } </pre>	

**Fig. 6.** The code on the top left is the standard C++ matrix multiply (ijk loop order) code. The code on the top right constructs a TaskGraph for the standard ijk matrix multiply loop. The code underneath shows an example of using the TaskGraph representation for the ijk matrix multiply kernel, together with SUIF-1 passes for interchanging and tiling loops to search for the optimal tilesize of the interchanged and tiled kernel for a particular architecture and problem size.

Figure 6 shows both the code for the standard C/C++ matrix multiply loop (ijk loop order) and the code for constructing a TaskGraph representing this loop, together with an example of how we can direct optimisations from the application program: we can interchange the for-j and for-k loops before compiling and executing the code. Further, we can perform loop tiling with a runtime-selected tile size. This last application demonstrates in particular the possibilities of using the TaskGraph library for domain-specific optimisation:

- *Optimising for a particular architecture*

In Fig. 6, we show a simple piece of code which implements a runtime search for the optimal tile size when tiling matrix multiply. In Fig. 8, we show the results of this search for both a Pentium 4-M (with 512K L2 cache) and an Athlon (with 256K L2 cache) processor. The resulting optimal tilesizes differ for most problem sizes, but they do not differ by as much as would have been expected if the optimal tile size was based on L2 capacity. We assume that a different parameter, such as TLB (translation lookaside buffer) span, is more significant in practice.

- *Optimising for a particular loop or working set*

The optimal tile size for matrix multiply calculated by our code shown in Fig. 6 differs across problem sizes (see Fig. 8). Similarly, we expect the optimal tile size to vary for different loop bodies and resulting working sets.

We believe that performance achieved with relatively straight-forward code in our matrix multiply example (up to 2 GFLOP/s on a Pentium 4-M 1.8 GHz, as shown in Fig. 7) illustrates the potential for the approach, but to achieve higher performance we would need to add hierarchical tiling, data copying to avoid associativity conflicts, unrolling and software pipelining; the ATLAS library [9], for comparison, achieves more than 4 GFLOPS on the Pentium 4-M above. Our approach allows applying at least some of the optimisations which are used by the ATLAS library to achieve its very high performance to TaskGraphs that have been written by the application programmer.

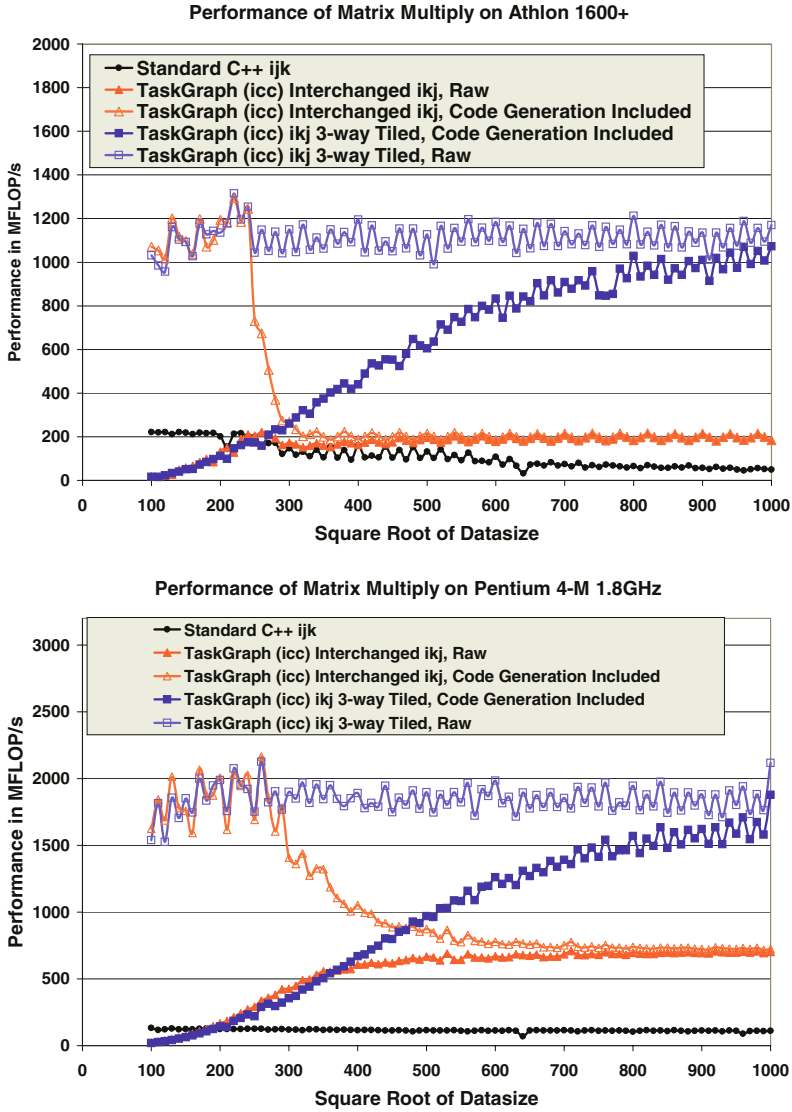
## 6 Related Work

In this section, we briefly discuss related work in the field of dynamic and multi-stage code optimisation.

### *Language-Based Approaches*

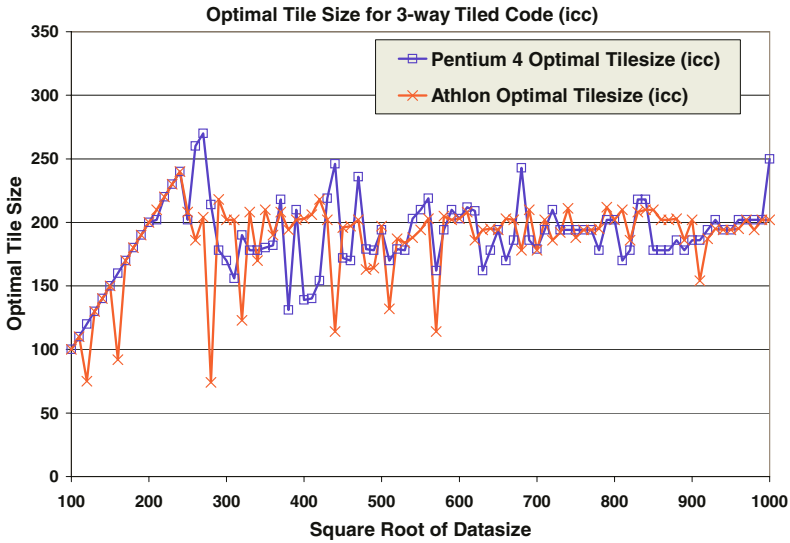
- *Imperative*

Tick-C or 'C [2], a superset of ANSI C, is a language for dynamic code generation. Like the TaskGraph library, 'C is explicit and imperative in nature; however, a key difference in the underlying design is that 'C relies on a special compiler (`gcc`). Dynamic code can be specified, composed and instantiated, i.e. compiled, at runtime. The fact that 'C relies on a special compiler also means that it is in some ways a more expressive and more powerful system than the TaskGraph library. For example, 'C facilitates the construction of dynamic function calls where the type and number of parameters is dynamically determined. This is not possible in the TaskGraph library. Jak [10], MetaML [11], MetaOCaml [12] and Template Haskell [13] are similar efforts, all relying on changes to the host language's syntax. Some of what we do with the TaskGraph library can be done using template metaprogramming in C++, which is used, for example, for loop fusion and temporary elimination in the Blitz++ and POOMA libraries [14, 15].



**Fig. 7.** Performance of single-precision matrix multiply on Athlon 1600+ with 256KB L2 cache and on Pentium 4-M 1.8 GHz with 512KB L2 cache. We show the performance of the naive C++ code (ijk loop order, compiled with gcc 3.3, -O3), the code where we have used the TaskGraph library to interchange the inner two loops (resulting in ikj loop order) and the code where the TaskGraph library is used to interchange and 3-way tile the loops. For the tiled code, we used the TaskGraph library to search for the optimal tile size for each data point, as shown in Fig. 6. For both the interchanged and tiled code, we plot one graph showing the raw performance of the generated code and one graph which shows the performance after the dynamic code generation cost has been amortised over one invocation of the generated code. All TaskGraph generated code was compiled with icc 7.1, -restrict -O3 -ipo -xiMKW -tpp7 -fno-alias.





**Fig. 8.** Optimal tile size on Athlon and Pentium 4-M processors, for each data point from Fig. 7. These results are based on a straight-forward exhaustive search implemented using the TaskGraph library’s runtime code restructuring capabilities (see code in Fig. 6).

– *Declarative*

DyC [3,5] is a dynamic compilation system which specialised selected parts of programs at runtime based on runtime information, such as values of certain data structures. DyC relies on declarative user annotations to trigger specialisation. This means that a sophisticated binding-time analysis is required which is both polyvariant (i.e. allowing specialisation of one piece of code for different combinations of static and dynamic variables) and program-point specific (i.e. allowing polyvariant specialisation to occur at arbitrary program points). The result of BTA is a set of *derived* static variables in addition to those variables which have been annotated as static. In order to reduce runtime compilation time, DyC produces, at compile-time, a *generating extension* [6] for each specialisation point. This is effectively a dedicated compiler which has been specialised to compile only the code which is being dynamically optimised. This static pre-planning of dynamic optimisation is referred as *staging*.

Marlet et al. [16] present a proposal for making the specialisation process itself more efficient. This is built using Tempo [17], an offline partial evaluator for C programs and also relies on an earlier proposal by Glück and Jørgensen to extend two-level binding-time analysis to multiple levels [18], i.e. to distinguish not just between dynamic and static variables but between multiple stages. The main contribution of Marlet et al. is to show that multi-level specialisation can be achieved more efficiently by repeated, incremental application of a two-level specialiser.

*Data-Flow Analysis.* Our library performs runtime data flow analysis on loops operating on arrays. A possible drawback with this solution could be high runtime overheads. Sharma et al. present deferred data-flow analysis (DDFA) [19] as a possible way of combining compile-time information with only limited runtime analysis in order to get accurate results. This technique relies on comprising the data flow information from *regions* of the control-flow graph into *summary functions*, together with a runtime *stitcher* which selects the applicable summary function, as well as computes summary function compositions at runtime.

*Transparent Dynamic Optimisation of Binaries.* One category of work on dynamic optimisation which contrasts with ours are approaches which do not rely on program source code but instead work in a transparent manner on running binaries. Dynamo [20] is a transparent dynamic optimisation system, implemented purely in software, which works on an executing stream of native instructions. Dynamo interprets the instruction stream until a *hot trace* of instructions is identified. This is then optimised, placed into a code cache and executed when the starting-point is re-encountered. These techniques also perform runtime code optimisation; however, as stated in Sec. 1, our objective is different: restructuring optimisation of software components with respect to context at runtime.

## 7 Ongoing and Future Work

We have recently evaluated the current TaskGraph library implementation in the context of some moderately large research projects [21]. This experience has led us to planning future developments of this work.

- *Automatic Generation of OpenMP Annotations*

We would like to use the runtime dependence information which is calculated by the TaskGraph library for automatically annotating the generated code with OpenMP [22] directives for SMP parallelisation. An alternative approach would be to use a compiler for compiling the generated code that has built-in SMP parallelisation capabilities.

- *Automatic Derivation of Component Metadata*

Our delayed evaluation, self-optimising (DESO) library of data-parallel numerical routines [23] currently relies on hand-written metadata which characterise the data placement constraints of components to perform cross-component data placement optimisation. One of the outstanding challenges which we would like to address in this work is to allow application programmers to write their own data-parallel components *without* having to understand and supply the placement-constraint metadata. We hope to generate these metadata automatically with the help of the TaskGraph library's dependence information. Some initial work on this project has been done [24].

- *Transparent Cross-Component Loop Fusion*

In an ongoing project [21] we are using the TaskGraph library to perform cross-component loop fusion in our DESO library of data-parallel numerical routines. This works by appending the taskgraphs representing each routine, then applying dependence analysis on adjacent loops to determine when fusion would be valid.

## 8 Conclusions and Discussion

We present the TaskGraph library as a tool for developing domain-specific optimisations in high-performance computing applications. The TaskGraph sub-language can be used in two distinct ways:

1. as an embedded language for constructing domain-specific components, which may be specialised to runtime requirements or data, and
2. as a component composition language, supporting applications which explicitly construct, optimise then execute composite computational plans.

In this volume, Lengauer [25] characterises four approaches to delivering domain-specific optimisations. Our goal is to offer a tool for building active libraries, which exploits his “two compilers” approach within a uniform and flexible framework.

The library illustrates the potential for embedding a domain-specific language in C++. The same technique could be used to embed languages for other domains, but we focused on classical scientific applications involving loops over arrays – in order to exploit the powerful techniques of restructuring and parallelising compiler research.

It is common to use template meta-programming in C++ to do this [26]. As surveyed by Czarnecki et al. in this volume [27], it is very attractive to design a language to support program generation as a first-class feature – as do, for example, Template Haskell and MetaOCaml.

We chose a dynamic, runtime approach (like MetaOCaml) to support our long-term goals of re-optimising software to adapt to changing context. This context may be the underlying CPU – as illustrated in Fig. 8, where we search the space of available transformations to best exploit the processor and memory system. We also aim to adapt to the shape and structure of changing data structures, such as a sparse matrix or an adaptive mesh. Similarly with resources, such as the number of processors or network contention. Finally, we’re interested in dynamic composition of components, for example in visualisation and data analysis.

The next step with this research – apart from applying it and encouraging others to do so – is to explore how to build domain-specific optimisations. The challenge here is to make building new optimisations easy enough for domain specialists. We need to build on (or replace) SUIF to support a rewriting-based approach to optimisation, as explored by Visser in this volume [28], and to facilitate extensions to the intermediate representation to support domain-specific program analyses. Of course, the real goal is to use such tools to extend our understanding of program optimisation.

## Acknowledgements

This work was supported by the United Kingdom EPSRC-funded OSCAR project (GR/R21486). We thank the referees for helpful and interesting comments.

## References

1. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices* **29** (1994) 31–37
2. Engler, D.R., Hsieh, W.C., Kaashoek, M.F.: 'C: a language for high-level, efficient, and machine-independent dynamic code generation. In: *POPL '96: Principles of Programming Languages*. (1996) 131–144
3. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science* **248** (2000) 147–199
4. McCarthy, J.: History of LISP. In: *The first ACM SIGPLAN Conference on History of Programming Languages*. Volume 13(8) of *ACM SIGPLAN Notices*. (1978) 217–223
5. Grant, B., Philipose, M., Mock, M., Chambers, C., Eggers, S.J.: An evaluation of staged run-time optimizations in DyC. In: *PLDI '99: Programming Language Design and Implementation*. (1999) 293–304
6. Jones, N.D.: Mix Ten Years Later. In: *PEPM '95: Partial Evaluation and Semantics-Based Program Manipulation*. (1995)
7. Intel Corporation: Integrated Performance Primitives for Intel Architecture. Reference Manual. Volume 2: Image and Video Processing. (200–2001)
8. Intel Corporation: Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual. (1999–2002) Available via [developer.intel.com](http://developer.intel.com).
9. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* **27** (2001) 3–35
10. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: *Fifth International Conference on Software Reuse*, IEEE Computer Society Press (1998) 143–153
11. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* **248** (2000) 211–242
12. Taha, W.: A gentle introduction to multi-stage programming (2004) In this volume.
13. Sheard, T., Peyton-Jones, S.: Template meta-programming for Haskell. *ACM SIGPLAN Notices* **37** (2002) 60–75
14. Veldhuizen, T.L.: Arrays in Blitz++. In: *ISCOPE'98: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments*. Number 1505 in *LNCS*, Springer-Verlag (1998) 223ff
15. Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W.J., Reynders, J., Smith, S., Williams, T.: Array design and expression evaluation in POOMA II. In: *ISCOPE'98: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments*. Number 231–238 in *LNCS* (1998) 223 ff
16. Marlet, R., Consel, C., Boinot, P.: Efficient incremental run-time specialization for free. *ACM SIGPLAN Notices* **34** (1999) 281–292 *Proceedings of PLDI'99*.
17. Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.N.: Tempo: Specializing systems applications and beyond. *ACM Computing Surveys* **30** (1998)
18. Glück, R., Jørgensen, J.: Fast binding-time analysis for multi-level specialization. In: *Perspectives of System Informatics*. Number 1181 in *LNCS* (1996) 261–272
19. Sharma, S., Acharya, A., Saltz, J.: Deferred Data-Flow Analysis. Technical Report TRCS98-38, University of California, Santa Barbara (1998)

20. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: A transparent dynamic optimization system. In: PLDI '00: Programming Language Design and Implementation. (2000) 1–12
21. Fordham, P.: Transparent run-time cross-component loop fusion. MEng Thesis, Department of Computing, Imperial College London (2002)
22. [www.opnemp.org](http://www.opnemp.org): OpenMP C and C++ Application Program Interface, Version 2.0 (2002)
23. Liniker, P., Beckmann, O., Kelly, P.H.J.: Delayed evaluation self-optimising software components as a programming model. In: Euro-Par 2002: Proceedings of the 8<sup>th</sup> International Euro-Par Conference. Number 2400 in LNCS (2002) 666–673
24. Subramanian, M.: A C++ library to manipulate parallel computation plans. Msc thesis, Department of Computing, Imperial College London, U.K. (2001)
25. Lengauer, C.: Program optimization in the domain of high-performance parallelism (2004) In this volume.
26. Veldhuizen, T.L.: C++ templates as partial evaluation. In: PEPM '99: Partial Evaluation and Semantic-Based Program Manipulation. (1999) 13–18
27. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL Implementation in MetaOCaml, Template Haskell, and C++ (2004) In this volume.
28. Visser, E.: Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9 (2004) In this volume.

# Guaranteed Optimization for Domain-Specific Programming

Todd L. Veldhuizen

Indiana University, Bloomington, Indiana 47401 USA  
[tveldhui@acm.org](mailto:tveldhui@acm.org)

**Abstract.** For software engineering reasons, it is often best to provide domain-specific programming environments in the context of a general-purpose language. In our view general-purpose languages are not yet general-purpose enough, and progress needs to be made before we can provide domain-specific languages that are both fast and safe. We outline some goals in this regard, and describe a possible implementation technology: *guaranteed optimization*, a technique for building compilers that provide proven guarantees of what optimizations they perform. Such optimizers can provide capabilities similar to staged languages, and thus provide the relevant performance improvements. They can also function as decision procedures, suggesting an approach of ‘optimizers as theorem provers,’ in which optimizing compilers can be used to check domain-specific safety properties and check proofs embedded in programs.

## 1 Introduction

There are several competing strategies for providing domain-specific programming environments: one can construct a wholly new language, extend an existing language, or work within an existing language. New languages are appealing because they’re fun to design and allow radical departures in syntax and semantics from existing languages; an example is logic-based program synthesis (e.g., [1]), in which purely declarative languages act as specifications for automatic programming. For non-programmers, a simple language that does exactly what they need can be less intimidating than general-purpose languages. However, new languages have drawbacks, too:

- New languages are hard to get right — semantics is a surprisingly tricky business.
- Users have to learn a new programming language, which can discourage adoption.
- Compilers require ongoing support to keep up with changing operating systems and architectures. One-off languages requiring special compilers are often research projects that founder when students graduate and professors move on to other interests.
- You can’t use features of multiple DSLs in one source file. For example, there currently exists a Fortran-like DSL that provides sparse arrays, and another that provides interval arithmetic. However, if one wants both sparse arrays and intervals, there is no compiler that supports both at once.

- One-off languages bring with them software-engineering risks, since they tend to lack the tool support (optimizing compilers, debuggers, development environments) of mainstream languages. To go down this road is to expose programmers to the prospect of a hundred thousand lines of misbehaving legacy code in an obscure language with no debugger.

Perhaps most importantly, one-off languages tend to lack many of the features one finds in mainstream languages, such as exceptions, language interoperability, threads, GUIs, and general-purpose I/O. Lacking these amenities, they can either languish in a limited niche, or their users will recreate what’s missing to make them more generally useful. There is a connection to be drawn between the feature creep that has turned Matlab, Perl, and Fortran into general-purpose languages and the process of creolization in natural languages. In creolization, pidgin languages developed for communication between two cultures (and here we might view Matlab as a “pidgin” of linear algebra notation and Basic) acquire native speakers and enlarge to become full-fledged languages. If people are taught a limited language, they will of necessity make it big enough to express everything that needs saying — no matter how awkwardly. In the New Guinea pidgin language Tok Pisin, which lacks the word *moustache*, the phrase *gras bilong maus* (grass belong mouth) conveys a visual impression of grass growing from someone’s lip. Such circumlocutions are akin to idioms developed by programming language communities to emulate missing language features. For example recursion, modules and dynamic memory management are all features absent from Fortran 77 and awkwardly emulated by users. These features were added to the language by Fortran 90. Instead of introducing a wholly new language which, if it becomes popular, is bound to evolve haphazardly toward general-purposeness, it makes sense to start with a general-purpose language powerful enough to encompass domain-specific uses.

So: unless one truly needs a language with radically new semantics or syntax, it’s prudent to work with a well-supported, general-purpose language (cf. [2]). Unfortunately the existing mainstream languages aren’t extensible enough to do interesting, high-performance DSL work in a natural way. Therefore a fruitful direction for research is to extend the reach of general-purpose languages; in this we pursue the old, probably unattainable dream of a universal language suitable for all purposes, a dream that goes back at least to Leibniz and his grandiose vision of a *Characteristica Universalis* in which to formalize all human thought (cf. [3]). In the past half-century useful connections have been found between logic, type theory, algebra, compilers, and verification; perhaps it is a good time to examine these threads of convergence and take stock of the prospects for universal programming languages.

**Structure of this paper.** This is a paper in two parts. In the first we describe hopes for general-purpose languages, which can be summarized as “pretty, fast, safe.” This provides background for the second half of the paper in which we propose technologies to make languages more general-purpose: guaranteed optimization, which addresses some aspects of performance, and proof embeddings, which address safety.

## 2 Pretty, Fast, Safe: Meditations on Extensible Languages

Aiming for universality means we should seek to span the gamut of needs, rather than targeting specific niches. Thus we begin with a broad survey of many different niches, setting our sights on one language to cover them all.

Our wish-list for a universal language can be summarized under these broad headings:

- **Pretty.** Programmers ought to be able to describe their problem in a form appropriate to their domain. We know that specialists invent new jargon and notations for their problem domains, and indeed that the development of notation is often central to gaining insight<sup>1</sup>.
- **Fast.** Performance should approach that of hand-crafted, low-level code, even for code written at high levels of abstraction.
- **Safe.** There should be mechanisms to statically check domain-specific safety properties.

In this paper we are concerned mostly with the *fast* and *safe* aspects, the pretty part — extensible syntax — having been studied in depth for decades, with macro systems, extensible parsers and the like. The problem of how to provide extensible languages with performance and safety, though, is a bit less well-explored.

**Fast.** With current compilers there is often a tradeoff between the expressiveness of code and its performance: code written close to the machine model will perform well, whereas code written at a high level of abstraction often performs poorly. This loss in performance associated with writing high-level code is often called the *abstraction penalty* [4–6]. The abstraction penalty can arise both from using a high level of abstraction, and also from syntax extensions (e.g., naive macro expansion). For this reason, compilers for *fast* extensible languages need to minimize this abstraction penalty. Traditional compiler optimizations are the obvious solution: disaggregation, virtual function elimination, and the like can greatly reduce the abstraction penalty. One of the shortcomings of current optimizing compilers is that they are unpredictable, with the result that performance-tuning is a fickle and frustrating art rather than a science. In the second half of this paper we propose a possible solution: optimizers that provide proven guarantees of what optimizations they will perform, thus making performance more predictable.

Beyond reducing the abstraction penalty, many problem areas admit *domain-specific optimizations* which can be exploited to achieve high performance. For example, operations on dense arrays admit an astonishing variety of performance improvements (e.g., [7]). A fundamental issue is whether domain-specific optimization is achieved by *transformation* or *generation* of code:

---

<sup>1</sup> “Some of the greatest advances in mathematics have been due to the invention of symbols, which it afterwards became necessary to explain; from the minus sign proceeded the whole theory of negative quantities.” – Aldous Huxley



- In transformation-based optimization one has low-level code (e.g., loop nests) that may represent high-level abstractions (e.g., array operations), and the optimizer seeks to recognize certain inefficient patterns and replace them with efficient code. Two difficulties with this approach are the *recognition problem* of finding pattern matches, and the fact that many interesting optimizations may violate the compiler’s notion of behavioural equivalence (thus, a good transformation may require one to play fast-and-loose with semantics, not a good thing.)
- In *generative optimization*, one starts from a high-level operation such as  $A = B + C * D$  and generates efficient code for it. This avoids both the recognition problem and taking liberties with semantics.

The transformation-based approach can be thought of as naive translation followed by smart optimization; conversely, the generative approach can be viewed as smart translation.

It’s not practical to include all such domain-specific optimizations in compilers, for economic reasons [8, 9]. Better, then, to package domain-specific optimizations with libraries; this is the basic idea behind *active libraries* [10]. Such libraries can capture performance-tuning expertise, providing better performance for average users, while still allowing experts to “roll-their-own.” This idea is similar in spirit to ongoing work by the parallel skeletons community (e.g., [11, 12]) who seek to capture patterns of parallel computing in reusable libraries of “skeletons.” There are many approaches to packaging optimizations and libraries together:

- “Meta-level” or “open compiler” systems let users manipulate syntax trees and other compiler data structures (e.g., [13–16]). While this technique provides enormous raw power, it does have drawbacks: obviously, letting users change compiler data structures raises some safety issues. Also, mainstream languages tend to be rather big, syntactically speaking, and so manipulating their syntax trees brings with it a certain complexity. It’s unclear how difficult this approach might be to implement for industrial languages since there are often many compilers for a given language, each having its own internal representations, and so one gets into messy issues of politics and economics.
- Annotations (e.g., [17]) can provide additional hints to the compiler about semantics and permissible transformations. While this approach is more limited in scope than metalevel approaches, it does avoid the complexity of manipulating syntax trees directly.
- Rewriting (e.g., [18–20]) is a promising technology for capturing domain-specific code transformations; it can express simple transforms elegantly. It’s not clear yet how cleanly one can express transforms that require some mixture of analysis and code generation.
- Staging (e.g., [21, 22]) and partial evaluation (e.g., [23]) lie somewhere between meta-level approaches and annotations in terms of power; they avoid the complexity of manipulating syntax trees while still allowing very useful

optimizations. Staging has been used to great effect in C++ to provide high-performance libraries for scientific computing, which we'll discuss shortly. In particular, staging and partial evaluation allow one to specialize code and (more generally) do component generation, which can have an enormous benefit to performance.

**Safe.** There are diverse (and firmly held!) ideas about what it means for a program to be safe. The question is what makes programs safe *enough*, absolute safety being unattainable in practice. The pragmatic approach is to recognize the existence a wide spectrum of safety levels, and that users should be free to choose a level appropriate to their purpose, avoiding a “one size fits all” mentality. At the low end of the spectrum are languages such as Matlab that do almost no static checking, deferring even syntactic checking of functions until they are invoked. At the other end is full-blown deductive verification that aims to prove correctness with respect to a specification. To aim for universality, one would like a single language capable of spanning this spectrum, and in particular to allow different levels of safety for different aspects of a program. For instance, it is common to check type-correctness statically, while deferring array bounds-checking until run time; this clearly represents two different standards of safety. Even program verification is typically applied only to some critical properties of a program, rather than attempting to exhaustively verify every aspect. Thus it is important to allow a mixture of safety levels within a single program.

Safety checks may be dynamic, as in (say) Scheme type checks [24] or Eiffel pre- and post-condition checking [25]. Dynamic checking can detect the presence of bugs at run-time; static checking can prove the absence of some bugs at compile-time. A notable difference between the two is that fully automatic static checking must necessarily reject some correct programs, due to the undecidability of most nontrivial safety conditions. Within static checking there is a wide range of ambition, from opportunistic bug-finding as in LCLint [26] and Metal [27], to lightweight verification of selected properties as in extended static checking [28] or SLAM [29], to full-blown deductive verification (e.g., PVS [30], Z [31], VDM [32]).

Systems that check programs using static analysis can be distinguished on the style of analysis performed: whether it is flow-, path-, or context-sensitive, for example. A recent trend has been to check static safety properties by shoe-horning them into the type system, for example checking locks [33, 34], array bounds [35], and security properties [36]. Type systems are generally flow- and context-invariant, whereas many of the static safety properties one would like to check are not. Thus it's not clear if type-based analyses are really the best way to go, since the approximations one gets are probably so coarse as to be of limited use in practice.

We can also distinguish between approaches in which safety checks are external to the program (as annotations, additional specifications, etc.) versus approaches in which safety checks are part of the program code (for example, run-time assertions, pre- and post-condition checking). Any artifact maintained

separately from the source code will tend to diverge from it, whether it be documentation, models or proofs. The “one-source” principle of software engineering suggests that safety checks should be integrated with the source code rather than separate from it. Some systems achieve this by placing annotations in the source code (e.g., ESC/Java [37]). However, this approach does not appear to integrate easily with staging; for example, can a stage produce customized safety checks for a later stage, when such checks are embedded in comments? Comments are not usually given staging semantics, so it’s unclear how this would work. Having safety checks *part of the language* ensures that they will interact with other language features (e.g., staging) in a sensible way.

Rather than relying solely on “external” tools such as model checkers or verification systems, we’d like as much as possible to have safety checking integrated with compilation. Why? First, for expedience: many checking tools rely on the same analyses required by optimization (points-to, alias, congruence), so it makes sense to combine these efforts. But our primary reason is that by integrating safety checking with compilers, we can provide libraries with the ability to perform their own static checks and emit customized diagnostics.

The approach we advocate is to define a general-purpose safety-checking system which subsumes both type-checking and domain-specific safety checks. Thus types are not given any special treatment, but rather treated the same as any other “domain-specific” safety property. Ideally one would also have extensible type systems, since many abstractions for problem domains have their own typing requirements. For example, deciding whether a tensor expression has a meaningful interpretation requires a careful analysis of indices and ranks. In scientific computing, dimension types (e.g., [38]) have been used to avoid mistakes such as assigning meters-per-second quantities to miles-per-hour variables. Having type checking handled by a general-purpose safety-checking system will likely open a way to extensible type systems.

Yet another aspect of safety checking is whether it is fully automatic (static analysis, model checking) or semi-automatic (e.g., proof assistants). There are many intriguing uses for a compiler supporting semi-automatic checking. By this we mean that a certain level of automated theorem proving takes place, but when checking fails, users can provide supplemental proofs of a property in order to proceed. This opens the way for libraries to issue *proof obligations* that must be satisfied by users (for example, to remove bound checks on arrays). Interesting safety properties are undecidable; this means any safety check must necessarily reject some safe programs. Thus, proof obligations would let users go beyond the limits of the compiler’s ability to automatically decide safety.

## 2.1 Reminiscences of a C++ Apostate

C++ has enjoyed success in providing domain-specific programming environments, and deserves attention as a case study. What distinguishes it is its staging mechanism (template metaprogramming), which has made possible the generation of highly efficient implementations for domain-specific abstractions. The key to this success is C++’s template mechanism, originally introduced to provide

parameterized types: one can create template classes such as `List<T>` where  $T$  is a type parameter, and instantiate it to particular instances such as `List<int>` and `List<string>`. (This idea was not new to C++ – a similar mechanism existed in Ada, and of course parametric polymorphism is a near cousin.) This instantiation involves duplicating the code and replacing the template parameters with their argument values – similar to polyvariant specialization in partial evaluation (c.f. [39]). In the development of templates it became clear that allowing dependent types such as `Vector<3>` would be useful (in C++ terminology called *non-type* template parameters); to type-check such classes it became necessary to evaluate expressions inside the `<>` brackets, so that `Vector<1 + 2>` is understood to be the same as `Vector<3>`. The addition of this evaluation step turned C++ into a staged language: arbitrary computations could be encoded as `<>`-expressions, which are guaranteed to be evaluated at compile-time. This capability was the basis of template metaprogramming and expression templates. A taste for these techniques is given by this definition of a function `pow` to calculate  $x^n$  (attribution unknown):

```
template<unsigned int N>
inline float pow(float x)
{ return pow<N % 2>(x) * pow<N / 2>(x*x); }

template<> inline float pow<1>(float x) { return x; }
template<> inline float pow<0>(float x) { return 1; }
```

The code `y=pow<5>(x)` expands at compile-time to something like `t1=x*x; t2=t1*t1; y=t2*x`. Using such techniques, C++ has been used to provide high-performance domain-specific libraries, for example POOMA [40] and Blitz [41] for dense arrays, MTL [42] and GMCL [43] for linear algebra. Blitz is able to do many of the dense array optimizations traditionally performed by compilers, such as loop fusion, interchange, and tiling. POOMA generates complicated parallel message-passing implementations of array operations from simple user code such as “`A=B+C*D`.” GMCL does elaborate component generation at compile-time, producing a concrete matrix type from a specification of element type, sparse or dense, storage format, and bounds checking provided by the user. Thus C++ is interesting because these libraries have been able to provide capabilities previously provided by optimizing compilers or component generation systems.

For syntactic abstraction, C++ provides only a fixed set of overloadable operators, each with a fixed precedence. This makes it a rather poor cousin of custom infix operators or general macro systems. However, people get surprising mileage out of this limited capability: one of the lessons learned from C++ is that the combination of staging and overloadable operators can be parlayed into customized parsing by turning a string of tokens into a tree and then traversing the tree in the compile-time stage to transform or ‘compile’ the expression.

Another useful lesson from the C++ experience is that staging can be used to provide static safety checks. Examples of this include ‘concept checking’ in C++ [44, 45], which essentially retrofits C++ with bounded polymorphism; SIUnits [46], which realizes the idea of dimension types [38] in C++ to check type safety

with respect to physical units; also, MTL and Blitz check conformance of matrices and arrays at compile-time. And as a more general example, we point to the `ctassert<>` template [47] which provides a compile-time analogue of dynamic `assert()` statements.

C++ was not intended to provide these features; they are largely serendipitous, the result of a flexible, general-purpose language design. Even now – a good decade after the basic features of C++ were laid down – people are still discovering novel (and useful!) techniques to accomplish things that were previously believed impossible; just recently it was discovered that compile-time reflection of arbitrary type properties was possible [48], something previously thought impossible. The lesson to be taken is that programming languages can have emergent properties; we mean “emergent properties” in the vague sense of surprising capabilities whose existence could not be foretold from the basic rules of the language. And here we must acknowledge Scheme, a small language that has proven amazingly capable due to such emergent properties. Such languages have an exuberance that makes them both fun and powerful; unanticipated features are bursting out all over! That emergent properties have proven so useful suggests that we try to foster languages with such potential. In the second half of this paper we describe our efforts in this direction.

### 3 Guaranteed Optimization and Proof Embeddings

The central technology in our approach to providing extensible languages is *guaranteed optimization*, which can be used to reduce abstraction penalty, remove overhead from naive expansion of syntax extensions, and provide staging-like capabilities. In the latter parts of this section we describe how guaranteed optimization can be used to prove theorems and check proofs embedded in programs.

#### 3.1 Optimizing Programs Thoroughly and Predictably

*Guaranteed Optimization* is a method for designing compilers that have proven guarantees of what optimizations they will perform. Here we give a taste for the method and explore its application to domain-specific programming environments; for a detailed explanation see [49, 50].

Guaranteed optimization is a “design-by-proof” technique: by attempting to prove a compiler has a certain property one uncovers failures in its design, and when at last the proof succeeds the compiler has the desired property. The inspiration comes from normal forms. Ideal optimizing compilers would compute normal forms of programs: every program would be reduced to some “optimal” program in its semantic equivalence class. This goal is of course out of reach, since program equivalence is undecidable; although we might be able to produce “optimal” programs for some classes of programs, we can’t do so in general for all programs. The best we can hope for is compilers that find normal forms of programs within some large, decidable subtheory of program equivalence, and this is what guaranteed optimization does.

A starting point is the goal that optimizing compilers should undo transformations we might apply to programs to make them “more abstract” for software-engineering purposes, for example replacing “ $1 + 2$ ” with

```
x = new Integer(1);
y = new Integer(2);
x.plus(y).intValue();
```

Not that anyone would write *that*, exactly, but similar code is routinely written for encapsulation and modularity. We can represent such transformations by rewrite rules. Some trivial examples of rewrites are:

```
R1.   $x \rightarrow x + 0$ 
R2.   $x \rightarrow \text{car}(\text{cons}(x, y))$ 
R3.   $x \rightarrow \text{if true then } x \text{ else } y$ 
      ⋮
```

These rules are, of course, completely backward from the usual approach: we work with rules that *de-optimize* a program. The obvious question is: why not devise an optimizer by reversing the direction of the arrows, using (say) Knuth-Bendix completion? The answer is that we work with an infinite number of such rules, some of which are conditional; thus we violate two requirements of Knuth-Bendix (which applies to a finite number of unconditional axioms.) Moreover, reversing the direction of the arrows can turn unconditional rewrites into conditional rewrites with undecidable conditions; and sensible approximations to the conditions require global analysis.

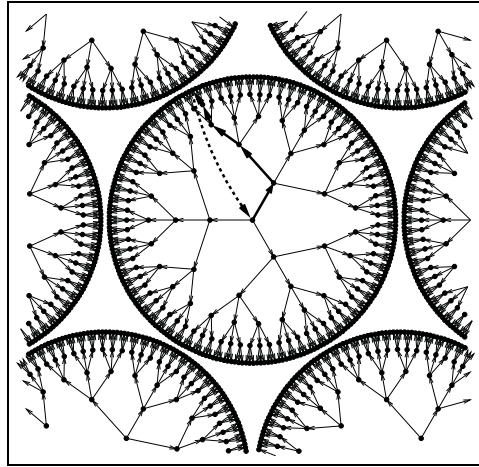
It turns out that the unusual approach of considering de-optimizing rules leads to a usable proof technique: we can prove that certain compilers undo any sequence of rule applications *in a single application of the optimizer*, yielding a program that is “minimally abstract.” Figure 1 illustrates.

In the next section we sketch the proof technique to give a taste for the methodology and results; for a detailed description see [49, 50].

**An overview of the proof technique.** In what follows, we write  $\rightarrow$  for the “de-optimizing” rewrite relation and  $x \leftrightarrow y$  (“convertible”) if  $x \rightarrow y$  or  $y \rightarrow x$ . We write  $\rightarrow^*$  and  $\leftrightarrow^*$  for the reflexive, transitive closure of  $\rightarrow$  and  $\leftrightarrow$ , respectively.

The proof technique applies to optimizers based on the *superanalysis* approach [51, 52]: instead of separate optimizing passes, there is instead a single, combined analysis followed by one transformation step. Superanalysis avoids the phase ordering problem of multiple optimization passes, a property crucial to the proof of guaranteed optimization. (As we discuss later, superanalysis also lets optimizers function as effective decision procedures in the presence of circularity e.g. recursion). Superanalysis-based optimizers consist of three steps:

1. The program text is examined to produce a system of analysis equations;
2. These equations are solved to a fixpoint;
3. Using this fixpoint solution, the program is transformed.



**Fig. 1.** A simplified sketch of Guaranteed Optimization. Each point represents a program and arrows indicate de-optimizing rewrites. The optimizer guarantees that it will undo any sequence of de-optimizing rewrites (bold arrows) in a single step (dashed arrow). Each circle represents a “behavioural equivalence class” of programs judged to have the same behaviour by the optimizer.

We can represent this process graphically for some program  $p$  as:

$$p \xrightarrow{(1) \text{ analysis}} \text{equations} \xrightarrow{(2)} \text{solution} \xrightarrow{(3)} \text{transformed program}$$

where the numbers (1), (2), (3) refer to the steps above. The essence of the proof technique is to consider a de-optimizing rewrite  $p \rightarrow p'$ , and compare what happens to both  $p$  and  $p'$  in each step of the optimizer:

$$\begin{array}{ccccccc}
 p & \xrightarrow{(1) \text{ analysis}} & \text{equations} & \xrightarrow{(2)} & \text{solution} & \xrightarrow{(3)} & \text{transformed program} \\
 \text{rewrite} \downarrow & & \vdots & & & & \parallel \\
 p' & \xrightarrow{(1) \text{ analysis}} & \text{equations} & \xrightarrow{(2)} & \text{solution} & \xrightarrow{(3)} & \text{transformed program}
 \end{array}$$

If the analysis is compositional – which for the optimizations we consider, it is – then a rewrite  $p \rightarrow p'$  can be viewed as inducing a rewrite on the analysis equations, shown above by a dotted line. (For example if one replaces  $x$  by  $x + 0$  in a program, there will be corresponding changes in the analysis equations to add new equations for 0 and  $x + 0$ .) By reasoning about this “induced rewrite” on the analysis equations one can prove properties of the solution that imply the transformed programs are equal (hence the double vertical line in the above figure.) It is convenient to think of the rewrite in terms of its context and redex: in a rewrite  $x + y \rightarrow x + (y + 0)$  we have the context  $x + [ ]$  and the redex  $y$ , where

[ ] denotes a *hole*. Thus the context is the portion of the program unchanged by the rewrite.

For each rewrite rule  $R1, R2, \dots$  one proves a lemma, a simplified example of which we give here for some rule  $R1$ :

**Lemma 1** *If  $p \rightarrow p'$  by rule  $R1$ , then (i) the analysis equations for  $p'$  have the same solution as the equations of  $p$  for program points in the rewrite context, and (ii) the transformed version of  $p'$  is the same as the transformed version of  $p$ .*

One proves a lemma of the above form for each rewrite considered. The effect of a rewrite on the fixpoint solution of a system of equations is reasoned about using the theory of fixpoint-preserving transformations and bisimilarity [53, 54]. We use these techniques to prove that analysis equations for program points in the context of  $p$  are bisimilar to those in  $p'$ , and thus the solution is unchanged for analysis variables in the context; and we prove enough about the solution to the  $p'$  equations to show that the additional code introduced by the rewrite is removed by the transformation step.

Proving these lemmas requires adjusting the design of the analysis and transformation so that the proof can succeed; thus, the proof technique is really a *design* technique for optimizing compilers. Once the lemmas are established, we can show that any sequence of rewrites is undone by the optimizer by a straightforward induction over rewrites, which we illustrate by stacking the above diagrams (Figure 2). Thus for any sequence of rewrites  $p \rightarrow p' \rightarrow p'' \rightarrow \dots$ , we have that the transformed versions of  $p, p', p'', \dots$  are all equal. Writing  $\mathcal{O} : \text{Program} \rightarrow \text{Program}$  for the optimizer, a guaranteed optimization proof culminates in a theorem of the form:

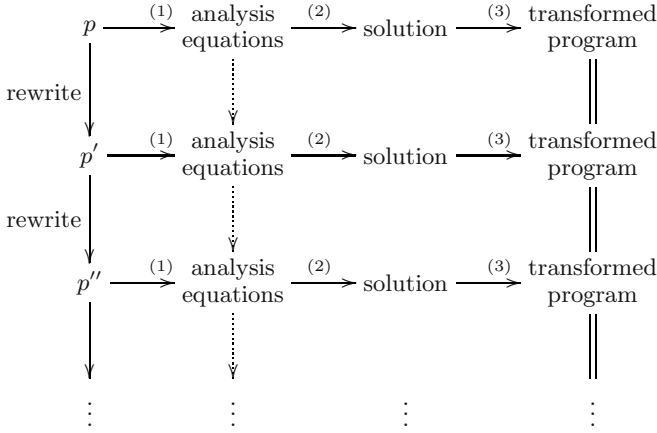
**Theorem 1 (Guaranteed Optimization)** *If  $p \xrightarrow{*} p'$ , then  $\mathcal{O}p = \mathcal{O}p'$ .*

This theorem simply states that any sequence of “de-optimizing” rewrites is undone by the optimizer. This theorem has some interesting implications. Recall that the kernel of  $\mathcal{O}$  is  $\ker \mathcal{O} = \{(p_1, p_2) \mid \mathcal{O}p_1 = \mathcal{O}p_2\}$ . If the optimizer is sound, then  $(p_1, p_2) \in \ker \mathcal{O}$  implies that  $p_1$  and  $p_2$  are behaviourally equivalent. Thus, one can view the optimizer  $\mathcal{O}$  as computing a normal form of programs with respect to a decidable subtheory of behavioural equivalence.

This notion of the kernel of an optimizer is generally useful, since it captures what we might call the “staging power” of a compiler; if  $\mathcal{O}_A$  and  $\mathcal{O}_B$  are two optimizing compilers and  $\ker \mathcal{O}_A \subseteq \ker \mathcal{O}_B$ , we can conclude that  $\mathcal{O}_B$  is a more powerful optimizer.

Another useful property one can prove with guaranteed optimization is the following: if one defines an *abstraction level*  $AL(p)$  of a program  $p$  as the length of the longest chain  $p_0 \rightarrow \dots \rightarrow p$ , then by imposing a few further requirements on the optimizer one can attain  $AL(\mathcal{O}p) = 0$ . Such optimizers find a minimal program with respect to the metric  $AL$ . Thus with appropriate “de-optimizing” rules  $\rightarrow$ , guaranteed optimization can address the abstraction penalty: optimized programs are guaranteed to be “minimally abstract” with respect to the rewrites  $\rightarrow$ .





**Fig. 2.** A diagram illustrating induction over rewrites: any sequence of de-optimizing rewrites is undone by the optimizer.

**Guaranteed optimization as staging.** Having described the basics of guaranteed optimization, we now examine its application to extensible compilation. In particular, we are interested in using guaranteed optimization to provide similar capabilities to staging and partial evaluation. We start by defining an evaluation relation  $\rightarrow$  sufficient to evaluate a purely functional subset of the intermediate language, for example:

- $E1.$  if true then  $e_1$  else  $e_2 \rightarrow e_1$
- $E2.$  if false then  $e_1$  else  $e_2 \rightarrow e_2$
- $E3.$   $n_1 + n_2 \rightarrow n_3$  where  $n_i \in \mathbb{Z}$ ,  $n_3 = n_1 + n_2$
- $\vdots$

Now, a good optimizer should clearly perform some of these evaluation steps at compile time by constant folding. We can use Guaranteed Optimization to design compilers that fully reduce a program with respect to  $\rightarrow$ . Suppose the compiler guarantees that applications of the “de-optimizing” rule  $R3$  are undone:

$$R3. \quad x \rightarrow \text{if true then } x \text{ else } y$$

Clearly the right-hand side of rule  $R3$  matches the left-hand side of rule  $E1$ ; so the optimizer will fully reduce a program with respect to  $E1$ . By making the set of “de-optimizing” rewrites  $\rightarrow$  big enough, in principle one can prove that:

$$\begin{array}{ccc} \rightarrow & \subseteq & \xrightarrow{*} \\ \text{(evaluation relation)} & & \text{(guaranteed optimization)} \end{array} \quad (1)$$

This becomes interesting and useful when the evaluation relation  $\rightarrow$  encompasses a Turing-complete subset of the language; then we have functionality closely

related to that of staging. In staging (e.g. [21, 22]; see also the chapter by Taha, this volume), one has evaluation relations such as:

$$\langle x + \sim(1 + 2) \rangle \twoheadrightarrow \langle x + 3 \rangle$$

Guaranteed optimization gives staging-like capability, but with fewer explicit annotations<sup>2</sup>:

$$x + (1 + 2) \twoheadrightarrow x + 3$$

So there is less of a syntactic burden to users, who no longer have to escape and defer obvious things; it is more like partial evaluation in this respect.

To summarize, guaranteed optimization can reduce the abstraction penalty; its staging-like capabilities can be used to remove code introduced by syntax extensions and to perform domain-specific optimizations.

### 3.2 Optimizers as Theorem Provers

To provide domain-specific safety checks, we need some kind of theorem proving capability. Many safety checks require the same analyses as optimization – points-to, congruence, and the like; the approach we’re exploring is to use the optimizer itself as an theorem-proving engine.

Sound program optimizers can be used to prove theorems: for example, if  $x$  is of type *int* and the optimizer replaces  $x + 0$  with  $x$ , then since the optimizer is sound  $x + 0 = x$  must hold in the theory of *int*.

We write  $\sim$  for behavioural equivalence on programs: if  $p_1 \sim p_2$ , then the behaviour of programs  $p_1$  and  $p_2$  are indistinguishable (Our working notion of behavioural equivalence is weak bisimilarity, with the observable behaviour of a program being its interaction with the operating system kernel and other processes via shared memory [55, 56]).

For the optimizer to be sound, we require that  $p \sim \mathcal{O}p$  – a program must be behaviourally equivalent to its optimized version. Therefore  $\mathcal{O}p = \mathcal{O}p'$  implies  $p \sim p'$ , or equivalently:

$$\ker \mathcal{O} \subseteq \sim \tag{2}$$

Thus one can view program optimizers as deciding a weaker behavioural equivalences on programs. Guaranteed optimization is, in essence, a proof that  $\ker \mathcal{O}$  satisfies certain closure properties related to the de-optimizing rewrites. From Theorem 1 (Guaranteed Optimization) we have  $p \xrightarrow{*} p' \Rightarrow \mathcal{O}p = \mathcal{O}p'$ . This implies  $\xrightarrow{*} \subseteq \ker \mathcal{O}$ ; together with soundness of  $\mathcal{O}$ , we have:

$$\xrightarrow{*} \subseteq \ker \mathcal{O} \subseteq \sim \tag{3}$$

The “de-optimizing” rewrites of guaranteed optimization can be viewed as oriented axioms. A compiler for which the guaranteed optimization proof succeeds

<sup>2</sup> To reach Turing-completeness of  $\twoheadrightarrow$  in guaranteed optimization, some annotations are still necessary to indicate which function applications are to be unfolded, and one must allow the possibility of nontermination of the optimizer, as in partial evaluation and staging.

is an effective decision procedure for the theory generated by these axioms (or, usually, a sound superset of the axioms). In related work [57] we describe a correspondence between the problem of how to effectively combine optimizing passes in a compiler, and how to combine decision procedures in an automated theorem prover. Rewrite-based or *pessimistic* optimizers can decide combinations of inductively defined theories in a manner similar to the Nelson-Oppen method of combining decision procedures [58]. On the other hand, optimizers based on *optimistic superanalysis*, of which guaranteed optimization is an example, can decide combinations of (more powerful) coinductively defined theories such as bisimulation.

This correspondence suggests using the compiler as a theorem prover, since the optimizer can prove properties of run-time values and behaviour. The optimizer fulfills a role similar to that of simplification engines in theorem provers: it can decide simple theorems on its own, for example  $x + 3 = 1 + 2 + x$  and  $\text{car}(\text{cons}(x, y)) = x$ . A number of interesting research directions are suggested:

- By including a language primitive `check(·)` that fails at compile-time if its argument is not provably true, one can provide a simple but crude version of domain-specific static checking, that would, in principle and assuming Eqn. (1), be at least be as good as static checks implementable with staging.
- In principle, one can embed a proof system  $\vdash$  in the language by encoding proof trees as objects, and rules as functions, such that a proof object is constructible in the language only if the corresponding  $\vdash$ -proof is. Such proofs can be checked by the optimizer; and deductions made by the optimizer (such as  $x + y = y + x$ ) can be used as premises for proofs. This is similar in spirit to the Curry-Howard isomorphism (e.g. [59]), although we embed proofs in values rather than types; and to proof-carrying code [60], although our approach is to intermingle proofs with the source code rather than having them separate.
- Such proof embeddings would let domain-specific libraries require *proof obligations* of users when automated static checks failed. For example, the expression `check( $x = y \vee P.\text{proves}(\text{equal}(x, y))$ )` would succeed only if  $x = y$  were proven by the optimizer, or if a proof object  $P$  were provided that gave a checkable proof of  $x = y$ .
- A reasonable test of a domain-specific safety checking system is whether it is powerful enough to subsume the type system. That is, one regards type checking as simply another kind of safety check to be implemented by a “type system library,” an approach we’ve previously explored in [61]. Embedded type systems are attractive because they open a natural route to user-extensible type systems, and hence domain-specific type systems.

## 4 A Summing Up

We have argued that making programming languages more general-purpose is a useful research direction for domain-specific programming environments. In our view, truly general-purpose languages should let libraries provide domain-specific

syntax extensions, performance improvements, and safety checks. Guaranteed optimization is an intriguing technology because it reduces abstraction penalty, can subsume staging, and its connections to theorem proving make it a good candidate for providing domain-specific safety checks.

The relationship between guaranteed optimization and other technologies providing staging-like abilities is summarized by this table:

	Annotation-free	Guarantees	Theorem Proving
Staging	○	●	○
Partial Evaluation	●	○	○
General Partial Computation	●	○	●
Guaranteed Optimization	◐	●	●

Staged languages (e.g., [21, 22]) are explicitly annotated with binding times, and have the advantage of guaranteeing evaluation of static computations at compile-time. The binding-time annotation is generally a *congruent division*, which effectively prevents theorem-proving about dynamic values.

Partial evaluation (e.g., [23]) automatically evaluates parts of a program at compile-time; in this respect it is closely related to guaranteed optimization. Partial evaluation is not usually understood to include proven guarantees of what will be statically evaluated; indeed, a lot of interesting research looks at effective heuristics for deciding what to evaluate. General partial computation [62] is an intriguing extension to partial evaluation in which theorem proving is used to reason about dynamic values.

Guaranteed optimization is largely annotation-free, although one must introduce some small annotations to control unfolding in the compile-time stage. It provides proven guarantees of what optimizations it will perform, and has the ability to prove theorems about run-time values.

## Acknowledgments

We thank Andrew Lumsdaine and the referees for helpful discussions and comments on drafts of this paper.

## References

1. Fischer, B., Visser, E.: Retrofitting the autobayes program synthesis system with concrete syntax (2004) In this volume.
2. Hudak, P.: Building domain-specific embedded languages. *ACM Computing Surveys* **28** (1996) 196–196
3. Thomas, W.: Logic for computer science: The engineering challenge. *Lecture Notes in Computer Science* **2000** (2001) 257–267
4. Stepanov, A.: Abstraction penalty benchmark (1994)
5. Robison, A.D.: The abstraction penalty for small objects in C++. In: POOMA’96: The Parallel Object-Oriented Methods and Applications Conference. (1996) Santa Fe, New Mexico.

6. Müller, M.: Abstraction benchmarks and performance of C++ applications. In: Proceedings of the Fourth International Conference on Supercomputing in Nuclear Applications. (2000)
7. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison Wesley, Reading, Mass. (1996)
8. Robison, A.D.: Impact of economics on compiler optimization. In: ISCOPE Conference on ACM 2001 Java Grande, ACM Press (2001) 1–10
9. Veldhuizen, T.L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98), SIAM Press (1999)
10. Czarnecki, K., Eisenecker, U., Glück, R., Vandevoorde, D., Veldhuizen, T.: Generative programming and active libraries (extended abstract). In Jazayeri, M., Musser, D., Loos, R., eds.: Generic Programming '98. Proceedings. Volume 1766 of Lecture Notes in Computer Science., Springer-Verlag (2000) 25–39
11. Botorog, G.H., Kuchen, H.: Efficient parallel programming with algorithmic skeletons. In L. Bouge, P. Fraigniaud, A. Mignotte, Y.R., ed.: Proceedings of EuroPar '96. Volume 1123 of LNCS., Springer (1996) 718–731
12. Küchen, H.: Optimizing sequences of skeleton calls (2004) In this volume.
13. Lamping, J., Kiczales, G., Rodriguez, L., Ruf, E.: An architecture for an open compiler. In Yonezawa, A., Smith, B.C., eds.: Proceedings of the International Workshop on New Models for Software Architecture '92: "Reflection and Meta-level Architecture". (1992)
14. Chiba, S.: A Metaobject Protocol for C++. In: OOPSLA'95. (1995) 285–299
15. Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., Maeda, M., Kubota, K.: Design and implementation of metalevel architecture in C++ – MPC++ approach. In: Reflection'96. (1996)
16. Engler, D.R.: Incorporating application semantics and control into compilation. In: USENIX Conference on Domain-Specific Languages (DSL'97). (October 15–17, 1997)
17. Guyer, S.Z., Lin, C.: An annotation language for optimizing software libraries. In: Domain-Specific Languages. (1999) 39–52
18. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. ACM SIGPLAN Notices **34** (1999) 13–26 Proceedings of the International Conference on Functional Programming (ICFP'98).
19. Elliott, C., Finne, S., de Moore, O.: Compiling embedded languages. In Taha, W., ed.: Semantics, Applications, and Implementation of Program Generation. Volume 1924 of Lecture Notes in Computer Science., Montreal, Springer-Verlag (2000) 9–27
20. Schupp, S., Gregor, D., Musser, D., Liu, S.M.: User-extensible simplification-type-based optimizer generators. In: Proceedings of Compiler Construction 2001. Volume 2027. (2001) 86–101
21. Nielson, F., Nielson, H.R.: Two-Level Functional Languages. Cambridge University Press, Cambridge, Mass. (1992)
22. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science **248** (2000) 211–242
23. Jones, N.D., Nielson, F.: Abstract interpretation. In Abramsky, S., Gabbay, D., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science. Volume 4. Oxford University Press (1995) To appear.
24. Kelsey, R., Clinger, W., (Editors), J.R.: Revised<sup>5</sup> report on the algorithmic language Scheme. ACM SIGPLAN Notices **33** (1998) 26–76

25. Meyer, B.: Eiffel: The Language. Prentice Hall (1991)
26. Evans, D., Gutttag, J., Horning, J., Tan, Y.: LCLint: a Tool for Using Specifications to Check Code. In Wile, D., ed.: Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering. Volume 19:5 of ACM SIGSOFT Software Engineering Notes., New Orleans, USA (1994) 87–96
27. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of Operating Systems Design and Implementation (OSDI). (2000)
28. Leino, K.R.M.: Extended static checking: A ten-year perspective. Lecture Notes in Computer Science **2000** (2001) 157–175
29. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2002) 1–3
30. Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: PVS: an experience report. In Hutter, D., Stephan, W., Traverso, P., Ullman, M., eds.: Applied Formal Methods—FM-Trends 98. Volume 1641 of Lecture Notes in Computer Science., Boppard, Germany, Springer-Verlag (1998) 338–345
31. Spivey, J.M.: The Z Notation: A Reference Manual. Second edn. International Series in Computer Sciences. Prentice-Hall, London (1992)
32. Jones, C.B.: Systematic Software Development Using VDM. Second edn. Prentice-Hall International, Englewood Cliffs, New Jersey (1990) ISBN 0-13-880733-7.
33. Flanagan, C., Abadi, M.: Types for safe locking. In: European Symposium on Programming. (1999) 91–108
34. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, ACM Press (2002) 211–230
35. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: SIGPLAN Conference on Programming Language Design and Implementation. (1998) 249–257
36. Volpano, D.M., Smith, G.: A type-based approach to program security. In: TAPSOFT. (1997) 607–621
37. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, ACM Press (2002) 234–245
38. Kennedy, A.: Dimension types. In Sannella, D., ed.: Programming Languages and Systems—ESOP’94, 5th European Symposium on Programming. Volume 788., Edinburgh, U.K., Springer (1994) 348–362
39. Veldhuizen, T.L.: C++ templates as partial evaluation. In Danvy, O., ed.: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. Technical report BRICS-NS-99-1, University of Aarhus, San Antonio, Texas, University of Aarhus, Dept. of Computer Science (1999) 13–18
40. Karmesin, S., Crottinger, J., Cummings, J., Haney, S., Humphrey, W., Reynnders, J., Smith, S., Williams, T.: Array design and expression evaluation in POOMA II. In: ISCOPE’98. Volume 1505., Springer-Verlag (1998) Lecture Notes in Computer Science.
41. Veldhuizen, T.L.: Arrays in Blitz++. In: Computing in Object-Oriented Parallel Environments: Second International Symposium (ISCOPE 98). Volume 1505 of Lecture Notes in Computer Science., Springer-Verlag (1998) 223–230

42. Siek, J.G., Lumsdaine, A.: The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In: International Symposium on Computing in Object-Oriented Parallel Environments. (1998)
43. Neubert, T.: Anwendung von generativen programmiertechniken am beispiel der matrixalgebra. Master's thesis, Technische Universität Chemnitz (1998) Diplomarbeit.
44. McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
45. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
46. Brown, W.E.: Applied template metaprogramming in SIUnits: The library of united-based computation. In: Second Workshop on C++ Template Programming. (2001)
47. Horn, K.S.V.: Compile-time assertions in C++. *C/C++ Users Journal* (1997)
48. Järvi, J., Willcock, J., Hinnant, H., Lumsdaine, A.: Function overloading based on arbitrary properties of types. *C/C++ Users Journal* **21** (2003) 25–32
49. Veldhuizen, T.L.: Active Libraries and Universal Languages. PhD thesis, Indiana University Computer Science (2004) (forthcoming).
50. Veldhuizen, T.L., Lumsdaine, A.: Guaranteed optimization: Proving nullspace properties of compilers. In: Proceedings of the 2002 Static Analysis Symposium (SAS'02). Volume 2477 of Lecture Notes in Computer Science., Springer-Verlag (2002) 263–277
51. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* **13** (1991) 181–210
52. Click, C., Cooper, K.D.: Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems* **17** (1995) 181–196
53. Wei, J.: Correctness of fixpoint transformations. *Theoretical Computer Science* **129** (1994) 123–142
54. Courcelle, B., Kahn, G., Vuillemin, J.: Algorithmes d'équivalence et de réduction à des expressions minimales dans une classe d'équations récursives simples. In Loeckx, J., ed.: *Automata, Languages and Programming*. Volume 14 of Lecture Notes in Computer Science., Springer Verlag (1974) 200–213
55. Milner, R.: *Communication and Concurrency*. International Series in Computer Science. Prentice Hall (1989)
56. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. *Theoretical Computer Science* **249** (2000) 3–80
57. Veldhuizen, T.L., Siek, J.G.: Combining optimizations, combining theories. Technical Report TR582, Indiana University Computer Science (2003)
58. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **1** (1979) 245–257
59. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard isomorphism. Technical report TOPPS D-368, Univ. of Copenhagen (1998)
60. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France (1997)
61. Veldhuizen, T.L.: Five compilation models for C++ templates. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
62. Futamura, Y., Nogi, K.: Generalized partial computation. In Bjørner, D., Ershov, A.P., Jones, N.D., eds.: *Proceedings of the IFIP Workshop on Partial Evaluation and Mixed Computation*, North-Holland (1987)

# Author Index

Batory, Don 1  
Beckmann, Olav 291  
Bischof, Holger 107

Consel, Charles 19, 165  
Cremet, Vincent 180  
Czarnecki, Krzysztof 51

Ertl, M. Anton 196

Fischer, Bernd 239

Gorlatch, Sergei 107, 274  
Gregg, David 196

Hammond, Kevin 127  
Houghton, Alastair 291

Kelly, Paul H.J. 291  
Kuchen, Herbert 254

Lengauer, Christian 73  
Leshchinskiy, Roman 107

Mellor, Michael 291  
Michaelson, Greg 127

O'Donnell, John T. 51, 143  
Odersky, Martin 180

Réveillère, Laurent 165

Smaragdakis, Yannis 92  
Striegnitz, Jörg 51

Taha, Walid 30, 51

Veldhuizen, Todd L. 307  
Visser, Eelco 216, 239